Table of Contents

1. Table of Contents ................................................................. i
2. Introduction ................................................................. 1
3. Architecture ................................................................. 3
4. Log4j 1.x Migration .......................................................... 10
5. API ................................................................. 17
6. Configuration ................................................................. 20
7. Web Applications and JSPs .................................................. 60
8. Plugins ................................................................. 69
9. Lookups ................................................................. 73
10. Appenders ................................................................. 81
11. Layouts ................................................................. 177
12. Filters ................................................................. 213
13. Async Loggers ............................................................ 228
14. Garbage-free Logging ..................................................... 242
15. JMX ................................................................. 251
16. Logging Separation ........................................................ 258
17. Extending Log4j ........................................................... 260
18. Programmatic Log4j Configuration ................................... 271
19. Custom Log Levels ........................................................ 279
1 Introduction

1.1 Welcome to Log4j 2!

1.1.1 Introduction

Almost every large application includes its own logging or tracing API. In conformance with this rule, the E.U. SEMPER project decided to write its own tracing API. This was in early 1996. After countless enhancements, several incarnations and much work that API has evolved to become log4j, a popular logging package for Java. The package is distributed under the Apache Software License, a fully-fledged open source license certified by the open source initiative. The latest log4j version, including full-source code, class files and documentation can be found at http://logging.apache.org/log4j/2.x/index.html.

Inserting log statements into code is a low-tech method for debugging it. It may also be the only way because debuggers are not always available or applicable. This is usually the case for multithreaded applications and distributed applications at large.

Experience indicates that logging was an important component of the development cycle. It offers several advantages. It provides precise context about a run of the application. Once inserted into the code, the generation of logging output requires no human intervention. Moreover, log output can be saved in persistent medium to be studied at a later time. In addition to its use in the development cycle, a sufficiently rich logging package can also be viewed as an auditing tool.

As Brian W. Kernighan and Rob Pike put it in their truly excellent book "The Practice of Programming":

As personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugging sessions are transient.

Logging does have its drawbacks. It can slow down an application. If too verbose, it can cause scrolling blindness. To alleviate these concerns, log4j is designed to be reliable, fast and extensible. Since logging is rarely the main focus of an application, the log4j API strives to be simple to understand and to use.

1.1.2 Log4j 2

Log4j 1.x has been widely adopted and used in many applications. However, through the years development on it has slowed down. It has become more difficult to maintain due to its need to be compliant with very old versions of Java and became End of Life in August 2015. Its alternative, SLF4J/Logback made many needed improvements to the framework. So why bother with Log4j 2?

Here are a few of the reasons.

1. Log4j 2 is designed to be usable as an audit logging framework. Both Log4j 1.x and Logback will lose events while reconfiguring. Log4j 2 will not. In Logback, exceptions in Appenders are never visible to the application. In Log4j 2 Appenders can be configured to allow the exception to percolate to the application.

2. Log4j 2 contains next-generation Asynchronous Loggers based on the LMAX Disruptor library. In multi-threaded scenarios Asynchronous Loggers have 10 times higher throughput and orders of magnitude lower latency than Log4j 1.x and Logback.
3. Log4j 2 is garbage free for stand-alone applications, and low garbage for web applications during steady state logging. This reduces pressure on the garbage collector and can give better response time performance.

4. Log4j 2 uses a Plugin system that makes it extremely easy to extend the framework by adding new Appenders, Filters, Layouts, Lookups, and Pattern Converters without requiring any changes to Log4j.

5. Due to the Plugin system configuration is simpler. Entries in the configuration do not require a class name to be specified.

6. Support for custom log levels. Custom log levels can be defined in code or in configuration.

7. Support for lambda expressions. Client code running on Java 8 can use lambda expressions to lazily construct a log message only if the requested log level is enabled. Explicit level checks are not needed, resulting in cleaner code.

8. Support for Message objects. Messages allow support for interesting and complex constructs to be passed through the logging system and be efficiently manipulated. Users are free to create their own Message types and write custom Layouts, Filters and Lookups to manipulate them.

9. Log4j 1.x supports Filters on Appenders. Logback added TurboFilters to allow filtering of events before they are processed by a Logger. Log4j 2 supports Filters that can be configured to process events before they are handled by a Logger, as they are processed by a Logger or on an Appender.

10. Many Logback Appenders do not accept a Layout and will only send data in a fixed format. Most Log4j 2 Appenders accept a Layout, allowing the data to be transported in any format desired.

11. Layouts in Log4j 1.x and Logback return a String. This resulted in the problems discussed at Logback Encoders. Log4j 2 takes the simpler approach that Layouts always return a byte array. This has the advantage that it means they can be used in virtually any Appender, not just the ones that write to an OutputStream.

12. The Syslog Appender supports both TCP and UDP as well as support for the BSD syslog and the RFC 5424 formats.

13. Log4j 2 takes advantage of Java 5 concurrency support and performs locking at the lowest level possible. Log4j 1.x has known deadlock issues. Many of these are fixed in Logback but many Logback classes still require synchronization at a fairly high level.

14. It is an Apache Software Foundation project following the community and support model used by all ASF projects. If you want to contribute or gain the right to commit changes just follow the path outlined at Contributing.
2 Architecture

2.1 Architecture

2.1.1 Main Components

Log4j uses the classes shown in the diagram below.

![Diagram of Log4j classes and their relationships]

Applications using the Log4j 2 API will request a Logger with a specific name from the LogManager. The LogManager will locate the appropriate LoggerContext and then obtain the Logger from it. If the Logger must be created, it will be associated with the LoggerConfig that contains either a) the same name as the Logger, b) the name of a parent package, or c) the root LoggerConfig. LoggerConfig objects are created from Logger declarations in the configuration. The LoggerConfig is associated with the Appenders that actually deliver the LogEvents.

2.1.1.1 Logger Hierarchy

The first and foremost advantage of any logging API over plain `System.out.println` resides in its ability to disable certain log statements while allowing others to print unhindered. This capability assumes that the logging space, that is, the space of all possible logging statements, is categorized according to some developer-chosen criteria.

In Log4j 1.x the Logger Hierarchy was maintained through a relationship between Loggers. In Log4j 2 this relationship no longer exists. Instead, the hierarchy is maintained in the relationship between LoggerConfig objects.

Loggers and LoggerConfigs are named entities. Logger names are case-sensitive and they follow the hierarchical naming rule:

**Named Hierarchy**
A LoggerConfig is said to be an ancestor of another LoggerConfig if its name followed by a dot is a prefix of the descendant logger name. A LoggerConfig is said to be a parent of a child LoggerConfig if there are no ancestors between itself and the descendant LoggerConfig.

For example, the LoggerConfig named "com.foo" is a parent of the LoggerConfig named "com.foo.Bar". Similarly, "java" is a parent of "java.util" and an ancestor of "java.util.Vector". This naming scheme should be familiar to most developers.

The root LoggerConfig resides at the top of the LoggerConfig hierarchy. It is exceptional in that it always exists and it is part of every hierarchy. A Logger that is directly linked to the root LoggerConfig can be obtained as follows:

```java
Logger logger = LogManager.getLogger(LogManager.ROOT_LOGGER_NAME);
```
Alternatively, and more simply:

```java
Logger logger = LogManager.getRootLogger();
```

All other Loggers can be retrieved using the `LogManager.getLogger` static method by passing the name of the desired Logger. Further information on the Logging API can be found in the Log4j 2 API.

2.1.1.2 LoggerContext

The `LoggerContext` acts as the anchor point for the Logging system. However, it is possible to have multiple active LoggerContexts in an application depending on the circumstances. More details on the LoggerContext are in the Log Separation section.

2.1.1.3 Configuration

Every LoggerContext has an active `Configuration`. The Configuration contains all the Appenders, context-wide Filters, LoggerConfigs and contains the reference to the StrSubstitutor. During reconfiguration two Configuration objects will exist. Once all Loggers have been redirected to the new Configuration, the old Configuration will be stopped and discarded.

2.1.1.4 Logger

As stated previously, Loggers are created by calling `LogManager.getLogger`. The Logger itself performs no direct actions. It simply has a name and is associated with a LoggerConfig. It extends AbstractLogger and implements the required methods. As the configuration is modified Loggers may become associated with a different LoggerConfig, thus causing their behavior to be modified.

2.2. Retrieving Loggers

Calling the `LogManager.getLogger` method with the same name will always return a reference to the exact same Logger object.

For example, in

```java
Logger x = LogManager.getLogger("wombat");
Logger y = LogManager.getLogger("wombat");
```

x and y refer to exactly the same Logger object.

Configuration of the log4j environment is typically done at application initialization. The preferred way is by reading a configuration file. This is discussed in Configuration.

Log4j makes it easy to name Loggers by software component. This can be accomplished by instantiating a Logger in each class, with the logger name equal to the fully qualified name of the
class. This is a useful and straightforward method of defining loggers. As the log output bears the name of the generating Logger, this naming strategy makes it easy to identify the origin of a log message. However, this is only one possible, albeit common, strategy for naming loggers. Log4j does not restrict the possible set of loggers. The developer is free to name the loggers as desired.

Since naming Loggers after their owning class is such a common idiom, the convenience method LogManager.getLogger() is provided to automatically use the calling class's fully qualified class name as the Logger name.

Nevertheless, naming loggers after the class where they are located seems to be the best strategy known so far.

2.1.1.5 LoggerConfig

LoggerConfig objects are created when Loggers are declared in the logging configuration. The LoggerConfig contains a set of Filters that must allow the LogEvent to pass before it will be passed to any Appenders. It contains references to the set of Appenders that should be used to process the event.

2. Log Levels

LoggerConfigs will be assigned a Log Level. The set of built-in levels includes TRACE, DEBUG, INFO, WARN, ERROR, and FATAL. Log4j 2 also supports custom log levels. Another mechanism for getting more granularity is to use Markers instead.

Log4j 1.x and Logback both have the concept of "Level Inheritance". In Log4j 2, Loggers and LoggerConfigs are two different objects so this concept is implemented differently. Each Logger references the appropriate LoggerConfig which in turn can reference its parent, thus achieving the same effect.

Below are five tables with various assigned level values and the resulting levels that will be associated with each Logger. Note that in all these cases if the root LoggerConfig is not configured a default Level will be assigned to it.

<table>
<thead>
<tr>
<th>Logger Name</th>
<th>Assigned LoggerConfig</th>
<th>LoggerConfig Level</th>
<th>Logger Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X.Y</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X.Y.Z</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
</tbody>
</table>

Example 1

In example 1 above, only the root logger is configured and has a Log Level. All the other Loggers reference the root LoggerConfig and use its Level.

<table>
<thead>
<tr>
<th>Logger Name</th>
<th>Assigned LoggerConfig</th>
<th>LoggerConfig Level</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>X.Y</td>
<td>X.Y</td>
<td>INFO</td>
<td>INFO</td>
</tr>
<tr>
<td>X.Y.Z</td>
<td>X.Y.Z</td>
<td>WARN</td>
<td>WARN</td>
</tr>
</tbody>
</table>
Example 2

In example 2, all loggers have a configured LoggerConfig and obtain their Level from it.

<table>
<thead>
<tr>
<th>Logger Name</th>
<th>Assigned LoggerConfig</th>
<th>LoggerConfig Level</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>X.Y</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>X.Y.Z</td>
<td>X.Y.Z</td>
<td>WARN</td>
<td>WARN</td>
</tr>
</tbody>
</table>

Example 3

In example 3, the loggers root, X and X.Y.Z each have a configured LoggerConfig with the same name. The Logger X.Y does not have a configured LoggerConfig with a matching name so uses the configuration of LoggerConfig X since that is the LoggerConfig whose name has the longest match to the start of the Logger's name.

<table>
<thead>
<tr>
<th>Logger Name</th>
<th>Assigned LoggerConfig</th>
<th>LoggerConfig Level</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>X.Y</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>X.Y.Z</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

Example 4

In example 4, the loggers root and X each have a Configured LoggerConfig with the same name. The loggers X.Y and X.Y.Z do not have configured LoggerConfigs and so get their Level from the LoggerConfig assigned to them, X, since it is the LoggerConfig whose name has the longest match to the start of the Logger's name.

<table>
<thead>
<tr>
<th>Logger Name</th>
<th>Assigned LoggerConfig</th>
<th>LoggerConfig Level</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>root</td>
<td>DEBUG</td>
<td>DEBUG</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>X.Y</td>
<td>X.Y</td>
<td>INFO</td>
<td>INFO</td>
</tr>
<tr>
<td>X.Y.Z</td>
<td>X</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

Example 5

In example 5, the loggers root, X, and X.Y each have a Configured LoggerConfig with the same name. The logger X.YZ does not have configured LoggerConfig and so gets its Level from the LoggerConfig assigned to it, X, since it is the LoggerConfig whose name has the longest match to the start of the Logger's name. It is not associated with LoggerConfig X.Y since tokens after periods must match exactly.
Example 6

In example 6, LoggerConfig X.Y it has no configured level so it inherits its level from LoggerConfig X. Logger X.Y.Z uses LoggerConfig X.Y since it doesn’t have a LoggerConfig with a name that exactly matches. It too inherits its logging level from LoggerConfig X.

The table below illustrates how Level filtering works. In the table, the vertical header shows the Level of the LogEvent, while the horizontal header shows the Level associated with the appropriate LoggerConfig. The intersection identifies whether the LogEvent would be allowed to pass for further processing (Yes) or discarded (No).

<table>
<thead>
<tr>
<th>Event Level</th>
<th>LoggerConfig Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>YES</td>
</tr>
<tr>
<td>TRACE</td>
<td>YES</td>
</tr>
<tr>
<td>DEBUG</td>
<td>YES</td>
</tr>
<tr>
<td>INFO</td>
<td>YES</td>
</tr>
<tr>
<td>WARN</td>
<td>YES</td>
</tr>
<tr>
<td>ERROR</td>
<td>YES</td>
</tr>
<tr>
<td>FATAL</td>
<td>YES</td>
</tr>
<tr>
<td>OFF</td>
<td>NO</td>
</tr>
</tbody>
</table>

2.1.1.6 Filter

In addition to the automatic log Level filtering that takes place as described in the previous section, Log4j provides Filters that can be applied before control is passed to any LoggerConfig, after control is passed to a LoggerConfig but before calling any Appenders, after control is passed to a LoggerConfig but before calling a specific Appender, and on each Appender. In a manner very similar to firewall filters, each Filter can return one of three results, Accept, Deny or Neutral. A response of Accept means that no other Filters should be called and the event should progress. A response of Deny means the event should be immediately ignored and control should be returned to the caller. A response of Neutral indicates the event should be passed to other Filters. If there are no other Filters the event will be processed.

Although an event may be accepted by a Filter the event still might not be logged. This can happen when the event is accepted by the pre-LoggerConfig Filter but is then denied by a LoggerConfig filter or is denied by all Appenders.
2.1.1.7 Appender

The ability to selectively enable or disable logging requests based on their logger is only part of the picture. Log4j allows logging requests to print to multiple destinations. In log4j speak, an output destination is called an Appender. Currently, appenders exist for the console, files, remote socket servers, Apache Flume, JMS, remote UNIX Syslog daemons, and various database APIs. See the section on Appenders for more details on the various types available. More than one Appender can be attached to a Logger.

An Appender can be added to a Logger by calling the addLoggerAppender method of the current Configuration. If a LoggerConfig matching the name of the Logger does not exist, one will be created, the Appender will be attached to it and then all Loggers will be notified to update their LoggerConfig references.

Each enabled logging request for a given logger will be forwarded to all the appenders in that Logger's LoggerConfig as well as the Appenders of the LoggerConfig's parents. In other words, Appenders are inherited additively from the LoggerConfig hierarchy. For example, if a console appender is added to the root logger, then all enabled logging requests will at least print on the console. If in addition a file appender is added to a LoggerConfig, say $C$, then enabled logging requests for $C$ and $C$’s children will print in a file and on the console. It is possible to override this default behavior so that Appender accumulation is no longer additive by setting additivity="false" on the Logger declaration in the configuration file.

The rules governing appender additivity are summarized below.

**Appender Additivity**

The output of a log statement of Logger $L$ will go to all the Appenders in the LoggerConfig associated with $L$ and the ancestors of that LoggerConfig. This is the meaning of the term "appender additivity".

However, if an ancestor of the LoggerConfig associated with Logger $L$, say $P$, has the additivity flag set to false, then $L$’s output will be directed to all the appenders in $L$’s LoggerConfig and it's ancestors up to and including $P$ but not the Appenders in any of the ancestors of $P$.

Loggers have their additivity flag set to true by default.

The table below shows an example:

<table>
<thead>
<tr>
<th>Logger Name</th>
<th>Added Appenders</th>
<th>Additivity Flag</th>
<th>Output Targets</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>A1</td>
<td>not applicable</td>
<td>A1</td>
<td>The root logger has no parent so additivity does not apply to it.</td>
</tr>
<tr>
<td>x</td>
<td>A-x1, A-x2</td>
<td>true</td>
<td>A1, A-x1, A-x2</td>
<td>Appenders of &quot;x&quot; and root.</td>
</tr>
<tr>
<td>x.y</td>
<td>none</td>
<td>true</td>
<td>A1, A-x1, A-x2</td>
<td>Appenders of &quot;x&quot; and root. It would not be typical to configure a Logger with no Appenders.</td>
</tr>
<tr>
<td>x.y.z</td>
<td>A-xyz1</td>
<td>true</td>
<td>A1, A-x1, A-x2, A-xyz1</td>
<td>Appenders in &quot;x.y.z&quot;, &quot;x&quot; and root.</td>
</tr>
</tbody>
</table>
2.1.1.8 Layout

More often than not, users wish to customize not only the output destination but also the output format. This is accomplished by associating a Layout with an Appender. The Layout is responsible for formatting the LogEvent according to the user’s wishes, whereas an appender takes care of sending the formatted output to its destination. The PatternLayout, part of the standard log4j distribution, lets the user specify the output format according to conversion patterns similar to the C language `printf` function.

For example, the PatternLayout with the conversion pattern "%r [%t] %-5p %c - %m%n" will output something akin to:
```
```

The first field is the number of milliseconds elapsed since the start of the program. The second field is the thread making the log request. The third field is the level of the log statement. The fourth field is the name of the logger associated with the log request. The text after the '-' is the message of the statement.

Log4j comes with many different Layouts for various use cases such as JSON, XML, HTML, and Syslog (including the new RFC 5424 version). Other appenders such as the database connectors fill in specified fields instead of a particular textual layout.

Just as importantly, log4j will render the content of the log message according to user specified criteria. For example, if you frequently need to log Oranges, an object type used in your current project, then you can create an OrangeMessage that accepts an Orange instance and pass that to Log4j so that the Orange object can be formatted into an appropriate byte array when required.

2.1.1.9 StrSubstitutor and StrLookup

The StrSubstitutor class and StrLookup interface were borrowed from Apache Commons Lang and then modified to support evaluating LogEvents. In addition the Interpolator class was borrowed from Apache Commons Configuration to allow the StrSubstitutor to evaluate variables that from multiple StrLookups. It too was modified to support evaluating LogEvents. Together these provide a mechanism to allow the configuration to reference variables coming from System Properties, the configuration file, the ThreadContext Map, StructuredData in the LogEvent. The variables can either be resolved when the configuration is processed or as each event is processed, if the component is capable of handling it. See Lookups for more information.
3 Log4j 1.x Migration

3.1 Migrating from Log4j 1.x

3.1.1 Using the Log4j 1.x bridge
Perhaps the simplest way to convert to using Log4j 2 is to replace the log4j 1.x jar file with Log4j 2's log4j-1.2-api.jar. However, to use this successfully applications must meet the following requirements:

1. They must not access methods and classes internal to the Log4j 1.x implementation such as Appenders, LoggerRepository or Category's callAppenders method.
2. They must not programmatically configure Log4j.
3. They must not configure by calling the classes DOMConfigurator or PropertyConfigurator.

3.1.2 Converting to the Log4j 2 API
For the most part, converting from the Log4j 1.x API to Log4j 2 should be fairly simple. Many of the log statements will require no modification. However, where necessary the following changes must be made.

1. The main package in version 1 is org.apache.log4j, in version 2 it is org.apache.logging.log4j
2. Calls to org.apache.log4j.Logger.getLogger() must be modified to org.apache.logging.log4j.LogManager.getLogger().
3. Calls to org.apache.log4j.Logger.getRootLogger() or org.apache.log4j.LogManager.getRootLogger() must be replaced with org.apache.logging.log4j.LogManager.getRootLogger().
4. Calls to org.apache.log4j.Logger.getLogger that accept a LoggerFactory must remove the org.apache.log4j.spi.LoggerFactory and use one of Log4j 2's other extension mechanisms.
5. Replace calls to org.apache.log4j.Logger.getEffectiveLevel() with org.apache.logging.log4j.Logger.getLevel().
6. Remove calls to org.apache.log4j.LogManager.shutdown(), they are not needed in version 2 because the Log4j Core now automatically adds a JVM shutdown hook on start up to perform any Core clean ups.
   1. Starting in Log4j 2.1, you can specify a custom ShutdownCallbackRegistry to override the default JVM shutdown hook strategy.
   2. Starting in Log4j 2.6, you can now use org.apache.logging.log4j.LogManager.shutdown() to initiate shutdown manually.
7. Calls to org.apache.log4j.Logger.setLevel() or similar methods are not supported in the API. Applications should remove these. Equivalent functionality is provided in the Log4j 2 implementation classes, see org.apache.logging.log4j.core.config.Configurator.setLevel(), but may leave the application susceptible to changes in Log4j 2 internals.
8. Where appropriate, applications should convert to use parameterized messages instead of String concatenation.
9. `org.apache.log4j.MDC` and `org.apache.log4j.NDC` have been replaced by the `Thread Context`.

### 3.1.3 Configuring Log4j 2

Although the Log4j 2 configuration syntax is different than that of Log4j 1.x, most, if not all, of the same functionality is available.

Note that system property interpolation via the `${foo}` syntax has been extended to allow property lookups from many different sources. See the [Lookups documentation](#) for more details. For example, using a lookup for the system property named `catalina.base`, in Log4j 1.x, the syntax would be `${catalina.base}`. In Log4j 2, the syntax would be `${sys:catalina.base}`.

Log4j 1.x has a `XMLLayout` which is different from the `XmlLayout` in Log4j 2, the `log4j-1.2-api` module contains a `Log4j1XmlLayout` which produce output in the format as in Log4j 1.x. The Log4j 1.x `SimpleLayout` can be emulated with `PatternLayout` "%level - %m%n". The Log4j 1.x `TTCCLayout` can be emulated with `PatternLayout` "%r [%t] %p %c %notEmpty{%ndc} - %m%n".

Both `PatternLayout` and `EnhancedPatternLayout` in Log4j 1.x can be replaced with `PatternLayout` in Log4j 2. The `log4j-1.2-api` module contains two pattern conversions "%ndc" and "%properties" which can be used to emulate "%x" and "%X" in Log4j 1.x `PatternLayout` ("%x" and "%X" in Log4j 2 have a slightly different format).

Below are the example configurations for Log4j 1.x and their counterparts in Log4j 2.

#### 3.1.3.1 Sample 1 - Simple configuration using a Console Appender

Log4j 1.x XML configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration PUBLIC "-//APACHE//DTD LOG4J 1.2//EN" "log4j.dtd">
<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
    </layout>
  </appender>
  <category name="org.apache.log4j.xml">
    <priority value="info"/>
  </category>
  <Root>
    <priority value="debug"/>
    <appender-ref ref="STDOUT"/>
  </Root>
</log4j:configuration>
```

Log4j 2 XML configuration

```xml
<log4j:configuration xmlns="http://log4j.org/xml/">
  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] %C{2} (%F:%L) - %m%n"/>
    </layout>
  </appender>
  <category name="org.apache.log4j.xml">
    <priority value="info"/>
  </category>
  <Root>
    <priority value="debug"/>
    <appender-ref ref="STDOUT"/>
  </Root>
</log4j:configuration>
```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
<Appenders>
<Console name="STDOUT" target="SYSTEM_OUT">
<PatternLayout pattern="%d %-5p [%t] %C{2} (%F:%L) - %m%n" /></Console>
</Appenders>
<Loggers>
<Logger name="org.apache.log4j.xml" level="info"/>
<Root level="debug">
<AppenderRef ref="STDOUT"/>
</Root>
</Loggers>
</Configuration>

3.1.3.2 Sample 2 - Simple configuration using a File Appender, XMLLayout and SimpleLayout

Log4j 1.x XML configuration

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration PUBLIC "-//APACHE//DTD LOG4J 1.2//EN" "log4j.dtd">
<appender name="A1" class="org.apache.log4j.FileAppender">
<param name="File" value="A1.log"/>
<param name="Append" value="false"/>
<layout class="org.apache.log4j.xml.XMLLayout"/>
</appender>
<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
<layout class="org.apache.log4j.SimpleLayout"/>
</appender>
<category name="org.apache.log4j.xml">
<priority value="debug"/>
</category>
<root>
<priority value="debug"/>
</root>
</log4j:configuration>

Log4j 2 XML configuration
3.1.3.3 Sample 3 - SocketAppender

Log4j 1.x XML configuration. This example from Log4j 1.x is misleading. The SocketAppender does not actually use a Layout. Configuring one will have no effect.

<?xml version="1.0" encoding="UTF-8"?>
<appender name="A1" class="org.apache.log4j.net.SocketAppender">
  <param name="RemoteHost" value="localhost"/>
  <param name="Port" value="5000"/>
  <param name="LocationInfo" value="true"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%t %-5p %c{2} - %m%n"/>
  </layout>
</appender>
<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
  </layout>
</appender>
<category name="org.apache.log4j.xml">
  <priority value="debug"/>
  <appender-ref ref="A1"/>
</category>
<root>
  <priority value="debug"/>
  <appender-ref ref="STDOUT"/>
</root>
</log4j:configuration>

Log4j 2 XML configuration
3.1.3.4 Sample 4 - AsyncAppender and TTCCLayout
Log4j 1.x XML configuration using the AsyncAppender.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" configDebug="true">
    <appender name="ASYNC" class="org.apache.log4j.AsyncAppender">
        <appender-ref ref="TEMP"/>
    </appender>
    <appender name="TEMP" class="org.apache.log4j.FileAppender">
        <param name="File" value="temp"/>
        <layout class="org.apache.log4j.TTCCLayout">
            <param name="ThreadPrinting" value="true"/>
            <param name="CategoryPrefixing" value="true"/>
            <param name="ContextPrinting" value="true"/>
        </layout>
    </appender>
    <root>
        <priority value="debug"/>
        <appender-ref ref="ASYNC"/>
    </root>
</log4j:configuration>
```

Log4j 2 XML configuration.
3.1.3.5 Sample 5 - AsyncAppender with Console and File
Log4j 1.x XML configuration using the AsyncAppender.

```xml
<Configuration status="debug">
  <Appenders>
    <File name="TEMP" fileName="temp">
      <PatternLayout pattern="%r [%t] %p %c %notEmpty{%ndc }- %m%n"/>
    </File>
    <Async name="ASYNC">
      <AppenderRef ref="TEMP"/>
    </Async>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="ASYNC"/>
    </Root>
  </Loggers>
</Configuration>
```

Log4j 2 XML configuration. Note that the Async Appender should be configured after the appenders it references. This will allow it to shutdown properly.
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug">
  <Appenders>
    <Console name="CONSOLE" target="SYSTEM_OUT">
      <PatternLayout pattern="\d \%5p \[\t\] \%C{2} \(\%F:\%L\) - \%m\n"/>
    </Console>
    <File name="TEMP" fileName="temp">
      <PatternLayout pattern="\d \%5p \[\t\] \%C{2} \(\%F:\%L\) - \%m\n"/>
    </File>
    <Async name="ASYNC">
      <AppenderRef ref="TEMP"/>
      <AppenderRef ref="CONSOLE"/>
    </Async>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="ASYNC"/>
    </Root>
  </Loggers>
</Configuration>
4.1 Log4j 2 API

4.1.1 Overview

The Log4j 2 API provides the interface that applications should code to and provides the adapter components required for implementers to create a logging implementation. Although Log4j 2 is broken up between an API and an implementation, the primary purpose of doing so was not to allow multiple implementations, although that is certainly possible, but to clearly define what classes and methods are safe to use in "normal" application code.

4.1.1.1 Hello World!

No introduction would be complete without the customary Hello, World example. Here is ours. First, a Logger with the name "HelloWorld" is obtained from the LogManager. Next, the logger is used to write the "Hello, World!" message, however the message will be written only if the Logger is configured to allow informational messages.

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class HelloWorld {
    private static final Logger logger = LogManager.getLogger("HelloWorld");

    public static void main(String[] args) {
        logger.info("Hello, World!");
    }
}
```

The output from the call to logger.info() will vary significantly depending on the configuration used. See the Configuration section for more details.

4.1.1.2 Substituting Parameters

Frequently the purpose of logging is to provide information about what is happening in the system, which requires including information about the objects being manipulated. In Log4j 1.x this could be accomplished by doing:

```java
if (logger.isDebugEnabled()) {
    logger.debug("Logging in user {} with birthday {}", user.getName(), user.getBirthdayCalendar());
}
```

Doing this repeatedly has the effect of making the code feel like it is more about logging than the actual task at hand. In addition, it results in the logging level being checked twice; once on the call to.isDebugEnabled and once on the debug method. A better alternative would be:

```java
logger.debug("Logging in user {} with birthday {}", user.getName(), user.getBirthdayCalendar());
```

With the code above the logging level will only be checked once and the String construction will only occur when debug logging is enabled.

4.1.1.3 Formatting Parameters

Formatter Loggers leave formatting up to you if toString() is not what you want. To facilitate formatting, you can use the same format strings as Java's Formatter. For example:
public static Logger logger = LogManager.getFormatterLogger("Foo");

logger.debug("Logging in user %s with birthday %s", user.getName(), user.getBirthDayCalendar());
logger.debug("Logging in user %1$s with birthday %2$tm %2$te,%2$tY", user.getName(), user.getBirthDayCalendar());
logger.debug("Integer.MAX_VALUE = %,d", Integer.MAX_VALUE);
logger.debug("Long.MAX_VALUE = %,d", Long.MAX_VALUE);

To use a formatter Logger, you must call one of the LogManager.getFormatterLogger methods. The output for this example shows that Calendar toString() is verbose compared to custom formatting:

2012-12-12 11:56:19,633 [main] DEBUG: Integer.MAX_VALUE = 2,147,483,647

4.1.1.4 Mixing Loggers with Formatter Loggers
Formatter loggers give fine-grained control over the output format, but have the drawback that the correct type must be specified (for example, passing anything other than a decimal integer for a %d format parameter gives an exception).

If your main usage is to use {}-style parameters, but occasionally you need fine-grained control over the output format, you can use the printf method:

public static Logger logger = LogManager.getLogger("Foo");

logger.debug("Opening connection to {}...", someDataSource);
logger.printf(Level.INFO, "Logging in user %1$s with birthday %2$tm %2$te,%2$tY", user.getName(), user.getBirthdayCalendar());

4.1.1.5 Java 8 lambda support for lazy logging
In release 2.4, the Logger interface added support for lambda expressions. This allows client code to lazily log messages without explicitly checking if the requested log level is enabled. For example, previously you would write:

```java
// pre-Java 8 style optimization: explicitly check the log level
// to make sure the expensiveOperation() method is only called if necessary
if (logger.isTraceEnabled()) {
    logger.trace("Some long-running operation returned {}", expensiveOperation());
}
```

With Java 8 you can achieve the same effect with a lambda expression. You no longer need to explicitly check the log level:

```java
// Java-8 style optimization: no need to explicitly check the log level:
// the lambda expression is not evaluated if the TRACE level is not enabled
logger.trace("Some long-running operation returned {}", () -> expensiveOperation());
```

4.1.1.6 Logger Names
Most logging implementations use a hierarchical scheme for matching logger names with logging configuration. In this scheme, the logger name hierarchy is represented by ‘.’ characters in the logger name, in a fashion very similar to the hierarchy used for Java package names. For example, org.apache.logging.append and org.apache.logging.filter both have org.apache.logging as their parent. In most cases, applications name their loggers by passing the
current class's name to `LogManager.getLogger(...)`. Because this usage is so common, Log4j 2 provides that as the default when the logger name parameter is either omitted or is null. For example, in all examples below the Logger will have a name of "org.apache.test.MyTest".

```java
package org.apache.test;

public class MyTest {
    private static final Logger logger = LogManager.getLogger(MyTest.class);
}

package org.apache.test;

public class MyTest {
    private static final Logger logger = LogManager.getLogger(MyTest.class.getName());
}

package org.apache.test;

public class MyTest {
    private static final Logger logger = LogManager.getLogger();
}
```
5 Configuration

5.1 Configuration

Inserting log requests into the application code requires a fair amount of planning and effort. Observation shows that approximately 4 percent of code is dedicated to logging. Consequently, even moderately sized applications will have thousands of logging statements embedded within their code. Given their number, it becomes imperative to manage these log statements without the need to modify them manually.

Configuration of Log4j 2 can be accomplished in 1 of 4 ways:

1. Through a configuration file written in XML, JSON, YAML, or properties format.
3. Programmatically, by calling the APIs exposed in the Configuration interface to add components to the default configuration.
4. Programmatically, by calling methods on the internal Logger class.

This page focuses primarily on configuring Log4j through a configuration file. Information on programmatically configuring Log4j can be found at Extending Log4j 2 and Programmatic Log4j Configuration.

Note that unlike Log4j 1.x, the public Log4j 2 API does not expose methods to add, modify or remove appenders and filters or manipulate the configuration in any way.

5.1.1 Automatic Configuration

Log4j has the ability to automatically configure itself during initialization. When Log4j starts it will locate all the ConfigurationFactory plugins and arrange them in weighted order from highest to lowest. As delivered, Log4j contains four ConfigurationFactory implementations: one for JSON, one for YAML, one for properties, and one for XML.

1. Log4j will inspect the "log4j.configurationFile" system property and, if set, will attempt to load the configuration using the ConfigurationFactory that matches the file extension.
2. If no system property is set the properties ConfigurationFactory will look for log4j2-test.properties in the classpath.
3. If no such file is found the YAML ConfigurationFactory will look for log4j2-test.yaml or log4j2-test.yml in the classpath.
4. If no such file is found the JSON ConfigurationFactory will look for log4j2-test.json or log4j2-test.jsn in the classpath.
5. If no such file is found the XML ConfigurationFactory will look for log4j2-test.xml in the classpath.
6. If a test file cannot be located the properties ConfigurationFactory will look for log4j2.properties on the classpath.
7. If a properties file cannot be located the YAML ConfigurationFactory will look for log4j2.yaml or log4j2.yml on the classpath.
8. If a YAML file cannot be located the JSON ConfigurationFactory will look for log4j2.json or log4j2.jsn on the classpath.
9. If a JSON file cannot be located the XML ConfigurationFactory will try to locate log4j2.xml on the classpath.
10. If no configuration file could be located the DefaultConfiguration will be used. This will cause logging output to go to the console.
An example application named MyApp that uses log4j can be used to illustrate how this is done.

```java
import com.foo.Bar;

// Import log4j classes.
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

public class MyApp {
    // Define a static logger variable so that it references the
    // Logger instance named "MyApp".
    private static final Logger logger = LogManager.getLogger(MyApp.class);

    public static void main(final String... args) {
        // Set up a simple configuration that logs on the console.
        logger.trace("Entering application.");
        Bar bar = new Bar();
        if (!bar.doIt()) {
            logger.error("Didn't do it.");
        }
        logger.trace("Exiting application.");
    }
}
```

MyApp begins by importing log4j related classes. It then defines a static logger variable with the name MyApp which happens to be the fully qualified name of the class.

MyApp uses the Bar class defined in the package com.foo.

```java
package com.foo;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

public class Bar {
    static final Logger logger = LogManager.getLogger(Bar.class.getName());

    public boolean doIt() {
        logger.entry();
        logger.error("Did it again!");
        return logger.exit(false);
    }
}
```

Log4j will provide a default configuration if it cannot locate a configuration file. The default configuration, provided in the DefaultConfiguration class, will set up:

- A `ConsoleAppender` attached to the root logger.
- A `PatternLayout` set to the pattern "\%d{HH:mm:ss.SSS} [\%t] %-5level \%logger{36} - \%msg \%n" attached to the `ConsoleAppender`

Note that by default Log4j assigns the root logger to `Level.ERROR`. 
The output of MyApp would be similar to:

17:13:01.540 [main] ERROR MyApp - Didn't do it.

As was described previously, Log4j will first attempt to configure itself from configuration files. A configuration equivalent to the default would look like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

Once the file above is placed into the classpath as log4j2.xml you will get results identical to those listed above. Changing the root level to trace will result in results similar to:

17:13:01.540 [main] ERROR MyApp - Didn't do it.

Note that status logging is disabled when the default configuration is used.

### 5.1.2 Additivity

Perhaps it is desired to eliminate all the TRACE output from everything except com.foo.Bar. Simply changing the log level would not accomplish the task. Instead, the solution is to add a new logger definition to the configuration:

```xml
<Logger name="com.foo.Bar" level="TRACE"/>
<Root level="ERROR">
  <AppenderRef ref="STDOUT"/>
</Root>
```

With this configuration all log events from com.foo.Bar will be recorded while only error events will be recorded from all other components.
In the previous example all the events from `com.foo.Bar` were still written to the Console. This is because the logger for `com.foo.Bar` did not have any appenders configured while its parent did. In fact, the following configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="com.foo.Bar" level="trace">
      <AppenderRef ref="Console"/>
    </Logger>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

would result in

```
17:13:01.540 [main] ERROR MyApp - Didn't do it.
```

Notice that the trace messages from `com.foo.Bar` appear twice. This is because the appender associated with logger `com.foo.Bar` writes the first instance to the Console. Next, the parent of `com.foo.Bar`, which in this case is the root logger, is referenced. The event is then passed to its appender, which also writes to the Console, resulting in the second instance. This is known as additivity. While additivity can be quite a convenient feature (as in the first previous example where no appender reference needed to be configured), in many cases this behavior is considered undesirable and so it is possible to disable it by setting the additivity attribute on the logger to false:
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="com.foo.Bar" level="trace" additivity="false">
      <AppenderRef ref="Console"/>
    </Logger>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>

Once an event reaches a logger with its additivity set to false the event will not be passed to any of its parent loggers, regardless of their additivity setting.

### 5.1.3 Automatic Reconfiguration

When configured from a File, Log4j has the ability to automatically detect changes to the configuration file and reconfigure itself. If the monitorInterval attribute is specified on the configuration element and is set to a non-zero value then the file will be checked the next time a log event is evaluated and/or logged and the monitorInterval has elapsed since the last check. The example below shows how to configure the attribute so that the configuration file will be checked for changes only after at least 30 seconds have elapsed. The minimum interval is 5 seconds.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration monitorInterval="30">
  ...
</Configuration>
```

### 5.1.4 Chainsaw can automatically process your log files (Advertising appender configurations)

Log4j provides the ability to 'advertise' appender configuration details for all file-based appenders as well as socket-based appenders. For example, for file-based appenders, the file location and the pattern layout in the file are included in the advertisement. Chainsaw and other external systems can discover these advertisements and use that information to intelligently process the log file.

The mechanism by which an advertisement is exposed, as well as the advertisement format, is specific to each Advertiser implementation. An external system which would like to work with a specific Advertiser implementation must understand how to locate the advertised configuration as well as the format of the advertisement. For example, a 'database' Advertiser may store configuration details in a database table. An external system can read that database table in order to discover the file location and the file format.
Log4j provides one Advertiser implementation, a 'multicastdns' Advertiser, which advertises appender configuration details via IP multicast using the http://jmdns.sourceforge.net library.

Chainsaw automatically discovers log4j's multicastdns-generated advertisements and displays those discovered advertisements in Chainsaw's Zeroconf tab (if the jmdns library is in Chainsaw's classpath). To begin parsing and tailing a log file provided in an advertisement, just double-click the advertised entry in Chainsaw's Zeroconf tab. Currently, Chainsaw only supports FileAppender advertisements.

To advertise an appender configuration:

• Add the JmDns library from http://jmdns.sourceforge.net to the application classpath
• Set the 'advertiser' attribute of the configuration element to 'multicastdns'
• Set the 'advertise' attribute on the appender element to 'true'
• If advertising a FileAppender-based configuration, set the 'advertiseURI' attribute on the appender element to an appropriate URI

FileAppender-based configurations require an additional 'advertiseURI' attribute to be specified on the appender. The 'advertiseURI' attribute provides Chainsaw with information on how the file can be accessed. For example, the file may be remotely accessible to Chainsaw via ssh/sftp by specifying a Commons VFS ( http://commons.apache.org/proper/commons-vfs/) sftp:// URI, an http:// URI may be used if the file is accessible through a web server, or a file:// URI can be specified if accessing the file from a locally-running instance of Chainsaw.

Here is an example advertisement-enabled appender configuration which can be used by a locally-running Chainsaw to automatically tail the log file (notice the file:// advertiseURI):

Please note, you must add the JmDns library from http://jmdns.sourceforge.net to your application classpath in order to advertise with the 'multicastdns' advertiser.

<?xml version="1.0" encoding="UTF-8"?>
<Configuration advertiser="multicastdns">
  ...
</Configuration>
<Appenders>
  <File name="File1" fileName="output.log" bufferedIO="false" advertiseURI="file://path/to/output.log" advertise="true">
    ...
  </File>
</Appenders>

5.1.5 Configuration Syntax

As of version 2.9, for security reasons, Log4j does not process DTD in XML files. If you want to split the configuration in multiple files, use XInclude or Composite Configuration.

As the previous examples have shown as well as those to follow, Log4j allows you to easily redefine logging behavior without needing to modify your application. It is possible to disable logging for certain parts of the application, log only when specific criteria are met such as the action being performed for a specific user, route output to Flume or a log reporting system, etc. Being able to do this requires understanding the syntax of the configuration files.

The configuration element in the XML file accepts several attributes:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>advertiser</td>
<td>(Optional) The Advertiser plugin name which will be used to advertise individual FileAppender or SocketAppender configurations. The only Advertiser plugin provided is &quot;multicastdns&quot;.</td>
</tr>
<tr>
<td>dest</td>
<td>Either &quot;err&quot; for stderr, &quot;out&quot; for stdout, a file path, or a URL.</td>
</tr>
<tr>
<td>monitorInterval</td>
<td>The minimum amount of time, in seconds, that must elapse before the file configuration is checked for changes.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the configuration.</td>
</tr>
<tr>
<td>packages</td>
<td>A comma separated list of package names to search for plugins. Plugins are only loaded once per classloader so changing this value may not have any effect upon reconfiguration.</td>
</tr>
<tr>
<td>schema</td>
<td>Identifies the location for the classloader to located the XML Schema to use to validate the configuration. Only valid when strict is set to true. If not set no schema validation will take place.</td>
</tr>
<tr>
<td>shutdownHook</td>
<td>Specifies whether or not Log4j should automatically shutdown when the JVM shuts down. The shutdown hook is enabled by default but may be disabled by setting this attribute to &quot;disable&quot;</td>
</tr>
<tr>
<td>shutdownTimeout</td>
<td>Specifies how many milliseconds appenders and background tasks will get to shutdown when the JVM shuts down. Default is zero which mean that each appender uses its default timeout, and don't wait for background tasks. Not all appenders will honor this, it is a hint and not an absolute guarantee that the shutdown procedure will not take longer. Setting this too low increase the risk of losing outstanding log events not yet written to the final destination. See LoggerContext.stop(long, java.util.concurrent.TimeUnit). (Not used if shutdownHook is set to &quot;disable&quot;).</td>
</tr>
<tr>
<td>status</td>
<td>The level of internal Log4j events that should be logged to the console. Valid values for this attribute are &quot;trace&quot;, &quot;debug&quot;, &quot;info&quot;, &quot;warn&quot;, &quot;error&quot; and &quot;fatal&quot;. Log4j will log details about initialization, rollover and other internal actions to the status logger. Setting status=&quot;trace&quot; is one of the first tools available to you if you need to troubleshoot log4j. (Alternatively, setting system property log4j2.debug will also print internal Log4j2 logging to the console, including internal logging that took place before the configuration file was found.)</td>
</tr>
<tr>
<td>strict</td>
<td>Enables the use of the strict XML format. Not supported in JSON configurations.</td>
</tr>
<tr>
<td>verbose</td>
<td>Enables diagnostic information while loading plugins.</td>
</tr>
</tbody>
</table>
5.1.6 Configuration with XML

Log4j can be configured using two XML flavors; concise and strict. The concise format makes configuration very easy as the element names match the components they represent however it cannot be validated with an XML schema. For example, the ConsoleAppender is configured by declaring an XML element named Console under its parent appenders element. However, element and attribute names are are not case sensitive. In addition, attributes can either be specified as an XML attribute or as an XML element that has no attributes and has a text value. So

```xml
<PatternLayout pattern="%m%n"/>
```

and

```xml
<PatternLayout>
  <Pattern>%m%n</Pattern>
</PatternLayout>
```

are equivalent.

The file below represents the structure of an XML configuration, but note that the elements in italics below represent the concise element names that would appear in their place.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Properties>
    <Property name="name1">value</property>
    <Property name="name2" value="value2"/>
  </Properties>
  <Appenders>
    <appender>
      <filter />
    </appender>
  </Appenders>
  <Loggers>
    <Logger name="name1">
      <filter />
    </Logger>
  </Loggers>
  <Root level="level">
    <AppenderRef ref="name"/>
  </Root>
</Configuration>
```

See the many examples on this page for sample appender, filter and logger declarations.
5. Strict XML

In addition to the concise XML format above, Log4j allows configurations to be specified in a more "normal" XML manner that can be validated using an XML Schema. This is accomplished by replacing the friendly element names above with their object type as shown below. For example, instead of the ConsoleAppender being configured using an element named Console it is instead configured as an appender element with a type attribute containing "Console".

```xml
<?xml version="1.0" encoding="UTF-8"?>;
<Configuration>
  <Properties>
    <Property name="name1">value</property>
    <Property name="name2" value="value2"/>
  </Properties>
  <Filter type="type" ... />
  <Appenders>
    <Appender type="type" name="name">
      <Filter type="type" ... />
    </Appender>
  </Appenders>
  ...
  <Logger name="name1">
    <Filter type="type" ... />
  </Logger>
  ...
  <Root level="level">
    <AppenderRef ref="name"/>
  </Root>
</Loggers>
</Configuration>

Below is a sample configuration using the strict format.
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" strict="true" name="XMLConfigTest"
    packages="org.apache.logging.log4j.test">
    <Properties>
        <Property name="filename">target/test.log</Property>
    </Properties>
    <Filter type="ThresholdFilter" level="trace"/>

    <Appenders>
        <Appender type="Console" name="STDOUT">
            <Layout type="PatternLayout" pattern="%m MDC%X%n"/>
            <Filters>
                <Filter type="MarkerFilter" marker="FLOW" onMatch="DENY" onMismatch="NEUTRAL"/>
                <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="DENY" onMismatch="ACCEPT"/>
            </Filters>
        </Appender>
        <Appender type="Console" name="FLOW">
            <Layout type="PatternLayout" pattern="%C{1}.%M %m %ex%n"/> <!-- class and line number -->
            <Filters>
                <Filter type="MarkerFilter" marker="FLOW" onMatch="ACCEPT" onMismatch="NEUTRAL"/>
                <Filter type="MarkerFilter" marker="EXCEPTION" onMatch="ACCEPT" onMismatch="DENY"/>
            </Filters>
        </Appender>
        <Appender type="File" name="File" fileName="${filename}">
            <Layout type="PatternLayout">
                <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
            </Layout>
        </Appender>
    </Appenders>

    <Loggers>
        <Logger name="org.apache.logging.log4j.test1" level="debug" additivity="false">
            <Filter type="ThreadContextMapFilter">
                <KeyValuePair key="test" value="123"/>
            </Filter>
            <AppenderRef ref="STDOUT"/>
        </Logger>
        <Logger name="org.apache.logging.log4j.test2" level="debug" additivity="false">
            <AppenderRef ref="File"/>
        </Logger>
        <Root level="trace">
            <AppenderRef ref="STDOUT"/>
        </Root>
    </Loggers>
</Configuration>
5.1.6.2 Configuration with JSON

In addition to XML, Log4j can be configured using JSON. The JSON format is very similar to the concise XML format. Each key represents the name of a plugin and the key/value pairs associated with it are its attributes. Where a key contains more than a simple value it itself will be a subordinate plugin. In the example below, ThresholdFilter, Console, and PatternLayout are all plugins while the Console plugin will be assigned a value of STDOUT for its name attribute and the ThresholdFilter will be assigned a level of debug.

```json
{
  "configuration": {
    "status": "error",
    "name": "RoutingTest",
    "packages": "org.apache.logging.log4j.test",
    "properties": {
      "property": {
        "name": "filename",
        "value": "target/rolling1/rollingtest-$${sd:type}.log" }
    },
    "ThresholdFilter": {
      "level": "debug" },
    "appenders": {
      "Console": { "name": "STDOUT",
      "PatternLayout": { "pattern": "\%m\n" },
      "ThresholdFilter": { "level": "debug" } },
      "Routing": { "name": "Routing",
      "Routes": {
        "pattern": "$${sd:type}" },
      "Route": [
        { "RollingFile": {
          "name": "Rolling-$${sd:type}" ,
          "filename": "$${filename}"
          "filePattern": "target/rolling1/test1-$${sd:type}.%i.log.gz",
          "PatternLayout": { "pattern": "\%d \%p \%c{1.} [\%t] \%m\n" },
          "SizeBasedTriggeringPolicy": { "size": "500" } }
        },
        { "AppenderRef": "STDOUT", "key": "Audit" } ]
      },
    "loggers": {
      "logger": { "name": "EventLogger", "level": "info", "additivity": "false",
      "AppenderRef": { "ref": "Routing" } },
      "root": { "level": "error", "AppenderRef": { "ref": "STDOUT" } }
    }
  }
}
```

Note that in the RoutingAppender the Route element has been declared as an array. This is valid because each array element will be a Route component. This won't work for elements such as appenders and filters, where each element has a different name in the concise format. Appenders and filters can be defined as array elements if each appender or filter declares an attribute named "type" that contains the type of the appender. The following example illustrates this as well as how to declare multiple loggers as an array.
{ "configuration": { "status": "debug", "name": "RoutingTest", "packages": "org.apache.logging.log4j.test", "properties": { "property": { "name": "filename", "value": "target/rolling1/rollingtest-${sd:type}.log" } }, "ThresholdFilter": { "level": "debug" }, "appenders": { "appender": [ { "type": "Console", "name": "STDOUT", "PatternLayout": { "pattern": "%m%n" }, "ThresholdFilter": { "type": "Routing", "name": "Routing", "Routes": [ { "pattern": "${sd:type}" }, "Route": [ { "RollingFile": { "name": "Rolling-${sd:type}" }, "filePattern": "target/rolling1/test1-${sd:type}.${i}.log.gz", "PatternLayout": { "pattern": "%d %p %C{1.} [%t] %m%n" }, "SizeBasedTriggeringPolicy": { "size": 500 } } ], "AppenderRef": "STDOUT", "key": "Audit" } ] }, "loggers": { "logger": [ { "name": "EventLogger", "level": "info", "additivity": "false", "AppenderRef": { "ref": "Routing" } }, { "name": "com.foo.bar", "level": "error", "additivity": "false", "AppenderRef": { "ref": "STDOUT" } } ], "root": { "level": "error", "AppenderRef": { "ref": "STDOUT" } } } ] }

Additional runtime dependencies are required for using JSON configuration files.

5.1.6.3 Configuration with YAML

Log4j also supports using YAML for configuration files. The structure follows the same pattern as both the XML and YAML configuration formats. For example:
Configuration:

status: warn
name: YAMLConfigTest
properties:
  property:
    name: filename
    value: target/test-yaml.log
thresholdFilter:
  level: debug
appenders:
  Console:
    name: STDOUT
    PatternLayout:
      Pattern: "%m%n"
  File:
    name: File
    fileName: ${filename}
    PatternLayout:
      Pattern: "%d %p %C{1.} [%t] %m%n"
    Filters:
      ThresholdFilter:
        level: error
Loggers:
  logger:
    -
      name: org.apache.logging.log4j.test1
      level: debug
      additivity: false
      ThreadContextMapFilter:
        KeyValuePair:
          key: test
          value: 123
      AppenderRef:
        ref: STDOUT
    -
      name: org.apache.logging.log4j.test2
      level: debug
      additivity: false
      AppenderRef:
        ref: File
Root:
  level: error
  AppenderRef:
    ref: STDOUT

Additional runtime dependencies are required for using YAML configuration files.
5.1.6.4 Configuration with Properties

As of version 2.4, Log4j now supports configuration via properties files. Note that the property syntax is NOT the same as the syntax used in Log4j 1. Like the XML and JSON configurations, properties configurations define the configuration in terms of plugins and attributes to the plugins.

Prior to version 2.6, the properties configuration requires that you list the identifiers of the appenders, filters and loggers, in a comma separated list in properties with those names. Each of those components will then be expected to be defined in sets of properties that begin with component.<identifier>.. The identifier does not have to match the name of the component being defined but must uniquely identify all the attributes and subcomponents that are part of the component. If the list of identifiers is not present the identifier must not contain a ‘.’. Each individual component MUST have a "type" attribute specified that identifies the component's Plugin type.

As of version 2.6, this list of identifiers is no longer required as names are inferred upon first usage, however if you wish to use more complex identifies you must still use the list. If the list is present it will be used.

Unlike the base components, when creating subcomponents you cannot specify an element containing a list of identifiers. Instead, you must define the wrapper element with its type as is shown in the policies definition in the rolling file appender below. You then define each of the subcomponents below that wrapper element, as the TimeBasedTriggeringPolicy and SizeBasedTriggeringPolicy are defined below.

Properties configuration files support the advertiser, monitorInterval, name, packages, shutdownHook, shutdownTimeout, status, verbose, and dest attributes. See Configuration Syntax for the definitions of these attributes.
status = error
dest = err
name = PropertiesConfig

property.filename = target/rolling/rollingtest.log

filter.threshold.type = ThresholdFilter
filter.threshold.level = debug

appender.console.type = Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %m%n
appender.console.filter.threshold.type = ThresholdFilter
appender.console.filter.threshold.level = error

appender.rolling.type = RollingFile
appender.rolling.name = RollingFile
appender.rolling.fileName = ${filename}
appender.rolling.fileName = $(filename)
appender.rolling.filePattern = target/rolling2/test1-%d{MM-dd-yy-HH-mm-ss}-%i.log.gz
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = %d %p %C{1.} [%t] %m%n
appender.rolling.policies.type = Policies
appender.rolling.policies.time.type = TimeBasedTriggeringPolicy
appender.rolling.policies.time.interval = 2
appender.rolling.policies.time.modulate = true
appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
appender.rolling.policies.size.size=100MB
appender.rolling.strategy.type = DefaultRolloverStrategy
appender.rolling.strategy.max = 5

logger.rolling.name = com.example.my.app
logger.rolling.level = debug
logger.rolling.additivity = false
logger.rolling.appenderRef.rolling.ref = RollingFile

rootLogger.level = info
rootLogger.appenderRef.stdout.ref = STDOUT

5.1.6.5 Configuring loggers
An understanding of how loggers work in Log4j is critical before trying to configure them. Please reference the Log4j architecture if more information is required. Trying to configure Log4j without understanding those concepts will lead to frustration.

A LoggerConfig is configured using the logger element. The logger element must have a name attribute specified, will usually have a level attribute specified and may also have an additivity attribute specified. The level may be configured with one of TRACE, DEBUG, INFO, WARN, ERROR, ALL or OFF. If no level is specified it will default to ERROR. The additivity attribute may be assigned a value of true or false. If the attribute is omitted the default value of true will be used.
A LoggerConfig (including the root LoggerConfig) can be configured with properties that will be added to the properties copied from the ThreadContextMap. These properties can be referenced from Appenders, Filters, Layouts, etc just as if they were part of the ThreadContextMap. The properties can contain variables that will be resolved either when the configuration is parsed or dynamically when each event is logged. See Property Substitution for more information on using variables.

The LoggerConfig may also be configured with one or more AppenderRef elements. Each appender referenced will become associated with the specified LoggerConfig. If multiple appenders are configured on the LoggerConfig each of them be called when processing logging events.

Every configuration must have a root logger. If one is not configured the default root LoggerConfig, which has a level of ERROR and has a Console appender attached, will be used. The main differences between the root logger and other loggers are

1. The root logger does not have a name attribute.
2. The root logger does not support the additivity attribute since it has no parent.

5.1.6.6 Configuring Appenders
An appender is configured either using the specific appender plugin's name or with an appender element and the type attribute containing the appender plugin's name. In addition each appender must have a name attribute specified with a value that is unique within the set of appenders. The name will be used by loggers to reference the appender as described in the previous section.

Most appenders also support a layout to be configured (which again may be specified either using the specific Layout plugin's name as the element or with "layout" as the element name along with a type attribute that contains the layout plugin's name. The various appenders will contain other attributes or elements that are required for them to function properly.

5.1.6.7 Configuring Filters
Log4j allows a filter to be specified in any of 4 places:

1. At the same level as the appenders, loggers and properties elements. These filters can accept or reject events before they have been passed to a LoggerConfig.
2. In a logger element. These filters can accept or reject events for specific loggers.
3. In an appender element. These filters can prevent or cause events to be processed by the appender.
4. In an appender reference element. These filters are used to determine if a Logger should route the event to an appender.

Although only a single filter element can be configured, that element may be the filters element which represents the CompositeFilter. The filters element allows any number of filter elements to be configured within it. The following example shows how multiple filters can be configured on the ConsoleAppender.
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" name="XMLConfigTest" packages="org.apache.logging.log4j.test">
  <Properties>
    <Property name="filename">target/test.log</Property>
  </Properties>
  <ThresholdFilter level="trace"/>
  <Appenders>
    <Console name="STDOUT">
      <PatternLayout pattern="%m MDC%X%n"/>
    </Console>
    <Console name="FLOW">
      <!-- this pattern outputs class name and line number -->
      <PatternLayout pattern="%C{1}.%M %m %ex%n"/>
      <filters>
        <MarkerFilter marker="FLOW" onMatch="ACCEPT" onMismatch="NEUTRAL"/>
        <MarkerFilter marker="EXCEPTION" onMatch="ACCEPT" onMismatch="DENY"/>
      </filters>
    </Console>
    <File name="File" fileName="${filename}">
      <PatternLayout>
        <pattern>%d %p %C{1.} [%t] %m%n</pattern>
      </PatternLayout>
    </File>
  </Appenders>
  <Loggers>
    <Logger name="org.apache.logging.log4j.test1" level="debug" additivity="false">
      <ThreadContextMapFilter>
        <KeyValuePair key="test" value="123"/>
      </ThreadContextMapFilter>
      <AppenderRef ref="STDOUT"/>
    </Logger>
    <Logger name="org.apache.logging.log4j.test2" level="debug" additivity="false">
      <Property name="user">${sys:user.name}</Property>
      <AppenderRef ref="File">
        <ThreadContextMapFilter>
          <KeyValuePair key="test" value="123"/>
        </ThreadContextMapFilter>
      </AppenderRef>
      <AppenderRef ref="STDOUT" level="error"/>
    </Logger>
    <Root level="trace">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
5.1.7 Property Substitution

Log4j 2 supports the ability to specify tokens in the configuration as references to properties defined elsewhere. Some of these properties will be resolved when the configuration file is interpreted while others may be passed to components where they will be evaluated at runtime. To accomplish this, Log4j uses variations of Apache Commons Lang's StrSubstitutor and StrLookup classes. In a manner similar to Ant or Maven, this allows variables declared as \${name} to be resolved using properties declared in the configuration itself. For example, the following example shows the filename for the rolling file appender being declared as a property.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" name="RoutingTest" packages="org.apache.logging.log4j.test">
    <Properties>
        <Property name="filename">target/rolling1/rollingtest-$${sd:type}.log</Property>
    </Properties>
    <ThresholdFilter level="debug"/>

    <Appenders>
        <Console name="STDOUT">
            <PatternLayout pattern="%m%n"/>
            <ThresholdFilter level="debug"/>
        </Console>
        <Routing name="Routing">
            <Routes pattern="$$[sd:type]">
                <Route>
                    <RollingFile name="Rolling-${sd:type}" fileName="${filename}" filePattern="target/rolling1/test1-${sd:type}.%i.log.gz">
                        <PatternLayout>
                            <pattern>%d %p %c{1.} [%t] %m%n</pattern>
                        </PatternLayout>
                        <SizeBasedTriggeringPolicy size="500"/>
                    </RollingFile>
                </Route>
                <Route ref="STDOUT" key="Audit"/>
            </Routes>
        </Routing>
    </Appenders>

    <Loggers>
        <Logger name="EventLogger" level="info" additivity="false">
            <AppenderRef ref="Routing"/>
        </Logger>

        <Root level="error">
            <AppenderRef ref="STDOUT"/>
        </Root>
    </Loggers>
</Configuration>
```

While this is useful, there are many more places properties can originate from. To accommodate this, Log4j also supports the syntax ${prefix:name} where the prefix identifies tells Log4j that variable
name should be evaluated in a specific context. See the Lookups manual page for more details. The contexts that are built in to Log4j are:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>Base64 encoded data. The format is ${base64:Base64_encoded_data}. For example: ${base64:SGVsbG8gV29ybGQhCg==} yields Hello World!</td>
</tr>
<tr>
<td>bundle</td>
<td>Resource bundle. The format is ${bundle:BundleName:BundleKey}. The bundle name follows package naming conventions, for example: ${bundle:com.domain.Messages:MyKey}</td>
</tr>
<tr>
<td>ctx</td>
<td>Thread Context Map (MDC)</td>
</tr>
<tr>
<td>date</td>
<td>Inserts the current date and/or time using the specified format</td>
</tr>
<tr>
<td>env</td>
<td>System environment variables. The formats are ${env:ENV_NAME} and ${env:ENV_NAME:-default_value}</td>
</tr>
<tr>
<td>jndi</td>
<td>A value set in the default JNDI Context.</td>
</tr>
<tr>
<td>jvmrunargs</td>
<td>A JVM input argument accessed through JMX, but not a main argument; see RuntimeMXBean.getInputArguments(). Not available on Android.</td>
</tr>
<tr>
<td>log4j</td>
<td>Log4j configuration properties. The expressions ${log4j:configLocation} and ${log4j:configParentLocation} respectively provide the absolute path to the log4j configuration file and its parent folder.</td>
</tr>
<tr>
<td>main</td>
<td>A value set with MapLookup.setMainArguments(String[])</td>
</tr>
<tr>
<td>map</td>
<td>A value from a MapMessage</td>
</tr>
<tr>
<td>sd</td>
<td>A value from a StructuredDataMessage. The key &quot;id&quot; will return the name of the StructuredDataId without the enterprise number. The key &quot;type&quot; will return the message type. Other keys will retrieve individual elements from the Map.</td>
</tr>
<tr>
<td>sys</td>
<td>System properties. The formats are ${sys:some.property} and ${sys:some.property:-default_value}.</td>
</tr>
</tbody>
</table>

### 5.1.8 Default Properties

A default property map can be declared in the configuration file by placing a Properties element directly after the Configuration element and before any Loggers, Filters, Appenders, etc. are declared. If the value cannot be located in the specified lookup the value in the default property map will be used. The default map is pre-populated with a value for "hostName" that is the current system's host
name or IP address and the "contextName" with is the value of the current logging context. See many places a Properties element is used in this section for examples.

### 5.1.9 Lookup Variables with Multiple Leading '\$' Characters

An interesting feature of StrLookup processing is that when a variable reference is declared with multiple leading '\$' characters each time the variable is resolved the leading '\$' is simply removed. In the previous example the "Routes" element is capable of resolving the variable at runtime. To allow this the prefix value is specified as a variable with two leading '\$' characters. When the configuration file is first processed the first 'S' character is simply removed. Thus, when the Routes element is evaluated at runtime it is the variable declaration "$\{sd:type\}" which causes the event to be inspected for a StructuredDataMessage and if one is present the value of its type attribute to be used as the routing key. Not all elements support resolving variables at runtime. Components that do will specifically call that out in their documentation.

If no value is found for the key in the Lookup associated with the prefix then the value associated with the key in the properties declaration in the configuration file will be used. If no value is found the variable declaration will be returned as the value. Default values may be declared in the configuration by doing:

```xml
<Configuration>
  <Properties>
    <Property name="type">Audit</Property>
  </Properties>
  ...
</Configuration>
```

As a footnote, it is worth pointing out that the variables in the RollingFile appender declaration will also not be evaluated when the configuration is processed. This is simply because the resolution of the whole RollingFile element is deferred until a match occurs. See [RoutingAppender](#) for more information.

### 5.1.10 Scripts

Log4j provides support for [JSR 223](#) scripting languages to be used in some of its components. Any language that provides support for the JSR 223 scripting engine may be used. A list of the languages and bindings for them can be found at the [Scripting Engine](#) web site. However, some of the languages listed there, such as JavaScript, Groovy and Beanshell, directly support the JSR 223 scripting framework and only require that the jars for that language be installed.

The components that support using scripts do so by allowing a `<script>`, `<scriptFile>`, or `<scriptRef>` element to be configured on them. The script element contains a name for the script, the language of the script, and the script text. The scriptFile element contains the name of the script, its location, its language, its charset, and whether the file should be watched for changes. The scriptRef element contains the name of the script that is defined in the `<scripts>` configuration element. The name of the script is used to store the script, along with its ScriptEngine, so it can quickly be located each time the script needs to be run. While the name is not required, providing it will help in debugging problems when the script is running. The language must be provided on the script element and must specify one of the language names that appear in the Configuration status log as described in the next section. If the language is not specified on the scriptFile element the language will be determined by
the file extension of the script path. If file monitoring is requested it will only be enabled if a non-zero monitorInterval is specified on the configuration element. That interval will be used to check for changes in the file.
<?xml version="1.0" encoding="UTF-8"?><Configuration status="debug" name="RoutingTest">
  <Scripts>
    <Script name="selector" language="javascript"><![CDATA[
      var result;
      if (logEvent.getLoggerName().equals("JavascriptNoLocation")) {
        result = "NoLocation";
      } else if (logEvent.getMarker() != null && logEvent.getMarker().isInstanceOf("FLOW")) {
        result = "Flow";
      }
      result;
    ]]>></Script>
    <ScriptFile name="groovy.filter" path="scripts/filter.groovy"/>
  </Scripts>
  <Appenders>
    <Console name="STDOUT">
      <ScriptPatternSelector defaultPattern="%d %p %m%n">
        <ScriptRef ref="selector"/>
        <PatternMatch key="NoLocation" pattern="%5level %c{1.} %msg%n"/>
        <PatternMatch key="Flow" pattern="%5level %c{1.} ====== %C{1.}.%M:%L %msg ======%n"/>
      </ScriptPatternSelector>
      <PatternLayout pattern="%m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="EventLogger" level="info" additivity="false">
      <ScriptFilter onMatch="ACCEPT" onMisMatch="DENY">
        <Script name="GroovyFilter" language="groovy"><![CDATA[
          if (logEvent.getMarker() != null && logEvent.getMarker().isInstanceOf("FLOW")) {
            return true;
          } else if (logEvent.getContextMap().containsKey("UserId")) {
            return true;
          }
          return false;
        ]]>></Script>
        <AppenderRef ref="STDOUT"/>
      </ScriptFilter>
    </Logger>
    <Root level="error">
      <ScriptFilter onMatch="ACCEPT" onMisMatch="DENY">
        <ScriptRef ref="groovy.filter"/>
      </ScriptFilter>
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
If the status attribute on the Configuration element is set to DEBUG the list of script engines currently installed and their attributes will be listed. Although some engines may say they are not thread safe, Log4j takes steps to insure that the scripts will run in a thread-safe manner if the engine advertises that it is not thread safe.

```java
var result;
if (logEvent.getLoggerName().equals("JavascriptNoLocation")) {
    result = "NoLocation";
} else if (logEvent.getMarker() != null && logEvent.getMarker().isInstanceOf("FLOW")) {
    result = "Flow";
}
result;
```

5.1.10.1 A special note on Beanshell
JSR 223 scripting engines are supposed to identify that they support the Compilable interface if they support compiling their scripts. Beanshell does this. However, whenever the compile method is called it throws an Error (not an Exception). Log4j catches this but will log the warning shown below for each Beanshell script when it tries to compile them. All Beanshell scripts will then be interpreted on each execution.
5.1.11 XInclude

XML configuration files can include other files with XInclude. Here is an example log4j2.xml file that includes two other files:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns:xi="http://www.w3.org/2001/XInclude"
 status="warn" name="XIncludeDemo">
 <properties>
  <property name="filename">xinclude-demo.log</property>
 </properties>
 <ThresholdFilter level="debug"/>
 <xi:include href="log4j-xinclude-appenders.xml" /> 
 <xi:include href="log4j-xinclude-loggers.xml" />
</configuration>
```

log4j-xinclude-appenders.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<appenders>
 <Console name="STDOUT">
  <PatternLayout pattern="%m%n" />
 </Console>
 <File name="File" fileName="${filename}" bufferedIO="true" immediateFlush="true">
  <PatternLayout>
   <pattern>%d %p %C{1.} [%t] %m%n</pattern>
  </PatternLayout>
 </File>
</appenders>
```

log4j-xinclude-loggers.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<loggers>
 <logger name="org.apache.logging.log4j.test1" level="debug" additivity="false">
  <ThreadContextMapFilter>
   <KeyValuePair key="test" value="123" /> 
  </ThreadContextMapFilter>
  <AppenderRef ref="STDOUT" />
 </logger>
 <logger name="org.apache.logging.log4j.test2" level="debug" additivity="false">
  <AppenderRef ref="File" />
 </logger>
 <root level="error">
  <AppenderRef ref="STDOUT" />
 </root>
</loggers>
```
5.1.12 Composite Configuration

Log4j allows multiple configuration files to be used by specifying them as a list of comma separated file paths on log4j.configurationFile. The merge logic can be controlled by specifying a class that implements the MergeStrategy interface on the log4j.mergeStrategy property. The default merge strategy will merge the files using the following rules:

1. The global configuration attributes are aggregated with those in later configurations replacing those in previous configurations, with the exception that the highest status level and the lowest monitorInterval greater than 0 will be used.
2. Properties from all configurations are aggregated. Duplicate properties replace those in previous configurations.
3. Filters are aggregated under a CompositeFilter if more than one Filter is defined. Since Filters are not named duplicates may be present.
4. Scripts and ScriptFile references are aggregated. Duplicate definitions replace those in previous configurations.
5. Appenders are aggregated. Appenders with the same name are replaced by those in later configurations, including all of the Appender’s subcomponents.
6. Loggers are all aggregated. Logger attributes are individually merged with duplicates being replaced by those in later configurations. Appender references on a Logger are aggregated with duplicates being replaced by those in later configurations. Filters on a Logger are aggregated under a CompositeFilter if more than one Filter is defined. Since Filters are not named duplicates may be present. Filters under Appender references included or discarded depending on whether their parent Appender reference is kept or discarded.

5.1.13 Status Messages

Troubleshooting tip for the impatient:

From log4j-2.9 onward, log4j2 will print all internal logging to the console if system property log4j2.debug is defined (with any or no value).

Prior to log4j-2.9, there are two places where internal logging can be controlled:

- Before a configuration is found, status logger level can be controlled with system property org.apache.logging.log4j.simplelog.StatusLogger.level.
- After a configuration is found, status logger level can be controlled in the configuration file with the "status" attribute, for example: <Configuration status="trace">

Just as it is desirable to be able to diagnose problems in applications, it is frequently necessary to be able to diagnose problems in the logging configuration or in the configured components. Since logging has not been configured, "normal" logging cannot be used during initialization. In addition, normal logging within appenders could create infinite recursion which Log4j will detect and cause the recursive events to be ignored. To accommodate this need, the Log4j 2 API includes a StatusLogger. Components declare an instance of the StatusLogger similar to:

protected final static Logger logger = StatusLogger.getLogger();

Since StatusLogger implements the Log4j 2 API's Logger interface, all the normal Logger methods may be used.

When configuring Log4j it is sometimes necessary to view the generated status events. This can be accomplished by adding the status attribute to the configuration element or a default value can be provided by setting the "Log4jDefaultStatusLevel" system property. Valid values of the status
attribute are "trace", "debug", "info", "warn", "error" and "fatal". The following configuration has the status attribute set to debug.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" name="RoutingTest">
  <Properties>
    <Property name="filename">target/rolling1/rollingtest-$${sd:type}.log</Property>
  </Properties>
  <ThresholdFilter level="debug"/>

  <Appenders>
    <Console name="STDOUT">
      <PatternLayout pattern="%m%n"/>
      <ThresholdFilter level="debug"/>
    </Console>
    <Routing name="Routing">
      <Routes pattern="$$\{sd:type\}">
        <Route>
          <RollingFile name="Rolling-$${sd:type}" fileName="$$\{filename\}" filePattern="target/rolling1/test1-$${sd:type}.%i.log.gz">
            <PatternLayout>
              <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <SizeBasedTriggeringPolicy size="500"/>
          </RollingFile>
        </Route>
        <Route ref="STDOUT" key="Audit"/>
      </Routes>
    </Routing>
  </Appenders>

  <Loggers>
    <Logger name="EventLogger" level="info" additivity="false">
      <AppenderRef ref="Routing"/>
    </Logger>
    <Root level="error">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

During startup this configuration produces:
2011-11-23 17:08:00,769 DEBUG Generated plugins in 0.003374000 seconds
2011-11-23 17:08:00,789 DEBUG Calling createProperty on class org.apache.logging.log4j.core.config.Property for element property with params(name="filename", value="target/rolling1/rollingtest-${sd:type}.log")
2011-11-23 17:08:00,792 DEBUG Calling configureSubstitutor on class org.apache.logging.log4j.core.config.plugins.PropertiesPlugin for element properties with params(properties={[filename=target/rolling1/rollingtest-${sd:type}.log])
2011-11-23 17:08:00,794 DEBUG Generated plugins in 0.001362000 seconds
2011-11-23 17:08:00,797 DEBUG Calling createFilter on class org.apache.logging.log4j.core.filter.ThresholdFilter for element ThresholdFilter with params(level="debug", onMatch="null", onMismatch="null")
2011-11-23 17:08:00,802 DEBUG Generated plugins in 0.001349000 seconds
2011-11-23 17:08:00,804 DEBUG Calling createAppender on class org.apache.logging.log4j.core.appender.ConsoleAppender for element Console with params(PatternLayout(%m%n), null, target="null", name="STDOUT", ignoreExceptions="null")
2011-11-23 17:08:00,804 DEBUG Calling createFilter on class org.apache.logging.log4j.core.filter.ThresholdFilter for element ThresholdFilter with params(level="debug", onMatch="null", onMismatch="null")
2011-11-23 17:08:00,813 DEBUG Calling createRoute on class org.apache.logging.log4j.core.appender.routing.Route for element Route with params(AppenderRef="null", key="null", Node=Route)
2011-11-23 17:08:00,823 DEBUG Calling createRoute on class org.apache.logging.log4j.core.appender.routing.Route for element Route with params(AppenderRef="STDOUT", key="Audit", Node=Route)
2011-11-23 17:08:00,825 DEBUG Calling createRoutes on class org.apache.logging.log4j.core.appender.routing.Routes for element Routes with params(pattern=${sd:type}, routes={Route(type=dynamic default), Route(type=static Reference=STDOUT key='Audit')})
2011-11-23 17:08:00,827 DEBUG Calling createAppender on class org.apache.logging.log4j.core.appender.routing.RoutingAppender for element Routing with params(name="Routing", ignoreExceptions="null", Routes={Route(type=dynamic default),Route(type=static Reference=STDOUT key='Audit')}), Configuration(RoutingTest), null, null)
2011-11-23 17:08:00,827 DEBUG Calling createAppenders on class org.apache.logging.log4j.core.config.plugins.AppendersPlugin for element appenders with params(appenders={STDOUT, Routing})
2011-11-23 17:08:00,828 DEBUG Calling createAppenderRef on class org.apache.logging.log4j.core.config.plugins.AppenderRefPlugin for element AppenderRef with params(ref="Routing")
2011-11-23 17:08:00,829 DEBUG Calling createLogger on class org.apache.logging.log4j.core.config.LoggerConfig for element logger with params(additivity=false, level="info", name="EventLogger", AppenderRef={Routing}, null)
2011-11-23 17:08:00,830 DEBUG Calling createAppenderRef on class org.apache.logging.log4j.core.config.plugins.AppenderRefPlugin for element AppenderRef with params(ref="STDOUT")
If the status attribute is set to error than only error messages will be written to the console. This makes troubleshooting configuration errors possible. As an example, if the configuration above is changed to have the status set to error and the logger declaration is:

```xml
<logger name="EventLogger" level="info" additivity="false">
  <AppenderRef ref="Routing"/>
</logger>
```

the following error message will be produced.


Applications may wish to direct the status output to some other destination. This can be accomplished by setting the dest attribute to either "err" to send the output to stderr or to a file location or URL. This can also be done by insuring the configured status is set to OFF and then configuring the application programmatically such as:

```java
StatusConsoleListener listener = new StatusConsoleListener(Level.ERROR);
StatusLogger.getLogger().registerListener(listener);
```
### 5.1.14 Testing in Maven

Maven can run unit and functional tests during the build cycle. By default, any files placed in `src/test/resources` are automatically copied to `target/test-classes` and are included in the classpath during execution of any tests. As such, placing a `log4j2-test.xml` into this directory will cause it to be used instead of a `log4j2.xml` or `log4j2.json` that might be present. Thus a different log configuration can be used during testing than what is used in production.

A second approach, which is extensively used by Log4j 2, is to set the `log4j.configurationFile` property in the method annotated with `@BeforeClass` in the `junit` test class. This will allow an arbitrarily named file to be used during the test.

A third approach, also used extensively by Log4j 2, is to use the `LoggerContextRule` JUnit test rule which provides additional convenience methods for testing. This requires adding the `log4j-core-test-jar` dependency to your test scope dependencies. For example:

```java
public class AwesomeTest {
    @Rule
    public LoggerContextRule init = new LoggerContextRule("MyTestConfig.xml");

    @Test
    public void testSomeAwesomeFeature() {
        final LoggerContext ctx = init.getLoggerContext();
        final Logger logger = init.getLogger("org.apache.logging.log4j.my.awesome.test.logger");
        final Configuration cfg = init.getConfiguration();
        final ListAppender app = init.getListAppender("List");
        logger.warn("Test message");
        final List<LogEvent> events = app.getEvents();
        // etc.
    }
}
```

### 5.1.15 System Properties

The Log4j documentation references a number of System Properties that can be used to control various aspects of Log4j 2 behavior. The table below lists these properties along with their default value and a description of what they control. Any spaces present in the property name are for visual flow and should be removed.

Note that beginning in Log4j 2.10, all system property names have been normalized to follow a consistent naming scheme. While the old property names are still supported for backwards compatibility, it is recommended to update configurations to use the new style. This system is extensible and is enabled through the `PropertySource` interface. Additional property source classes can be added through the standard `ServiceLoader` mechanism in Java SE.

Properties can be overridden by sources with a lower number priority (e.g., -100 comes before 100). The following sources are all available by default:

<table>
<thead>
<tr>
<th>Source</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Environment Variables

Environment variables are all prefixed with LOG4J_, are in all caps, and words are all separated by underscores. Only this naming scheme is supported for environment variables as there were no old naming schemes to maintain compatibility with.

log4j2.component.properties file

Including this file on the classpath can be used as an alternative to providing properties as system properties. This has priority over system properties, but they can be overridden by environment variables as described above.

System Properties

All properties can be set using normal system property patterns. These have the lowest priority and can be overridden by included properties files or environment variables.

PropertySource priorities and descriptions

The following is a list of available global configuration properties. Note that these can only be set once per JVM process unlike configuration settings available in configuration files. The Property Name column contains the name used in properties files and system properties; Environment Variable for the equivalent environment variable; and Legacy Property Name for the pre-2.10 name.

<table>
<thead>
<tr>
<th>Property Name (Legacy Property Name)</th>
<th>Environment Variable</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log4j2.configurationFile (log4j2.configurationFile)</td>
<td>LOG4J_CONFIGURATION</td>
<td>Path to an Log4j 2 configuration file. May also contain a comma separated list of configuration file names.</td>
<td></td>
</tr>
<tr>
<td>log4j2.debug (log4j2.debug)</td>
<td>LOG4J_DEBUG</td>
<td>Log4j2 will print all internal logging to the console if system property log4j2.debug is defined (with any or no value).</td>
<td></td>
</tr>
<tr>
<td>log4j2.mergeFactory (log4j2.mergeFactory)</td>
<td>LOG4J_MERGE_FACTORY</td>
<td>The name of the class that implements the MergeStrategy interface. If not specified DefaultMergeStrategy will be used when creating a CompositeConfiguration..</td>
<td></td>
</tr>
</tbody>
</table>
### Configuration

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log4j2.contextSelector</code></td>
<td>Creates the <code>LoggerContext</code> class. An application can have one or more active <code>LoggerContexts</code> depending on the circumstances. See Log Separation for more details. Available context selector implementation classes:</td>
</tr>
<tr>
<td><code>ClassLoaderContextSelector</code></td>
<td>- separate <code>LoggerContexts</code> for each web application.</td>
</tr>
<tr>
<td><code>JndiContextSelector</code></td>
<td>- use JNDI to locate each web application's <code>LoggerContext</code>.</td>
</tr>
<tr>
<td><code>BundleContextSelector</code></td>
<td>- separate <code>LoggerContexts</code> for each OSGi bundle.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log4j2.logEventFactory</code></td>
<td>Factory class used by <code>LoggerConfig</code> to create <code>LogEvent</code> instances. (Ignored when the <code>AsyncLoggerContextSelector</code> is used.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log4j2.loggerContextFactory</code></td>
<td>Factory class used by <code>LogManager</code> to bootstrap the logging implementation. The core jar provides</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log4j2.configurationFactory</code></td>
<td>Fully specified class name of a class extending</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log4j2.shutdownHookEnabled</td>
<td>Overrides the global flag for whether or not a shutdown hook should be used to stop a LoggerContext. By default, this is enabled and can be disabled on a per-configuration basis. When running with the log4j-web module, this is automatically disabled.</td>
</tr>
<tr>
<td>log4j2.shutdownCallbackRegistry</td>
<td>Fully specified class name of a class implementing ShutdownCallbackRegistry. If specified, an instance of this class is used instead of DefaultShutdownCallbackRegistry. The specified class must have a default constructor.</td>
</tr>
<tr>
<td>log4j2.clock</td>
<td>Implementation of the org.apache.logging.log4j.core.Clock interface that is used for timestamping the log events. By default, System.currentTimeMillis is called on every log event. You can also specify a fully qualified class name of a custom class that implements the Clock interface.</td>
</tr>
<tr>
<td>log4j2.level</td>
<td>Log level of the default configuration. The default configuration is used if the ConfigurationFactory could not successfully create a configuration (e.g. no log4j2.xml file was found).</td>
</tr>
<tr>
<td>log4j2.disableThreadContext</td>
<td>If true, the ThreadContext stack and map are disabled. (May be ignored if a custom ThreadContext map is specified.)</td>
</tr>
<tr>
<td>log4j2.disableThreadContextStack</td>
<td>If true, the ThreadContext stack is disabled.</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>log4j2.disableThreadContextMap</code></td>
<td>If true, the ThreadContext map is disabled. (May be ignored if a custom ThreadContext map is specified.)</td>
</tr>
<tr>
<td><code>log4j2.threadContextMap</code></td>
<td>Fully specified class name of a custom ThreadContextMap implementation class.</td>
</tr>
<tr>
<td><code>log4j2.isThreadContextMapInheritable</code></td>
<td>If true use a InheritableThreadLocal to implement the ThreadContext map. Otherwise, use a plain ThreadLocal. (May be ignored if a custom ThreadContext map is specified.)</td>
</tr>
<tr>
<td><code>log4j2.contextDataInjector</code></td>
<td>Fully specified class name of a custom ContextDataInjector implementation class.</td>
</tr>
<tr>
<td><code>log4j2.garbagefreeThreadContextMap</code></td>
<td>Specify &quot;true&quot; to make the ThreadContext map garbage-free.</td>
</tr>
<tr>
<td><code>log4j2.disableJmx</code></td>
<td>If true, Log4j configuration objects like LoggerContexts, Appenders, Loggers, etc. will not be instrumented with MBeans and cannot be remotely monitored and managed.</td>
</tr>
<tr>
<td><code>log4j2.jmxNotifyAsync</code></td>
<td>If true, log4j's JMX notifications are sent from a separate background thread, otherwise they are sent from the caller thread. If system property <code>log4j2.is.webapp</code> is true or the <code>javax.servlet.Servlet</code> class is on the classpath, the default behaviour is to use the caller thread to send JMX notifications.</td>
</tr>
<tr>
<td><code>log4j2.skipJansi</code></td>
<td>If true, the ConsoleAppender will not try to use the Jansi output stream on Windows.</td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>log4j2.ignoreTCL</td>
<td>(log4j.ignoreTCL) false</td>
</tr>
<tr>
<td>log4j2.uuidSequence</td>
<td>LOG4J_UUID_SEQUENCE 0</td>
</tr>
<tr>
<td>log4j2.simplelogShowContext</td>
<td>LOG4J_SIMPLELOG_SHO false</td>
</tr>
<tr>
<td>log4j2.simplelogShowLogname</td>
<td>LOG4J_SIMPLELOG_SHO false</td>
</tr>
<tr>
<td>log4j2.simplelogShowShortLogname</td>
<td>LOG4J_SIMPLELOG_SHO true</td>
</tr>
<tr>
<td>log4j2.simplelogShowDateTime</td>
<td>LOG4J_SIMPLELOG_SHO false</td>
</tr>
<tr>
<td>log4j2.simplelogLogFile</td>
<td>LOG4J_SIMPLELOG_LOG system.err</td>
</tr>
<tr>
<td>log4j2.simplelogLevel</td>
<td>LOG4J_SIMPLELOG_LEVEL ERROR</td>
</tr>
<tr>
<td>log4j2.simplelog.&lt;loggerName&gt;.level</td>
<td>LOG4J_SIMPLELOG_.&lt;loggerName&gt;.level</td>
</tr>
</tbody>
</table>
log4j2.simplelogStatusLogger

LOG4J_SIMPLELOG_STATUS_LOGGER

ERROR

This property is used to control the initial StatusLogger level, and can be overridden in code by calling

StatusLogger.getLogger().setLevel(someLevel).

Note that the StatusLogger level is only used to determine the status log output level until a listener is registered. In practice, a listener is registered when a configuration is found, and from that point onwards, status messages are only sent to the listeners (depending on their statusLevel).

log4j2.defaultStatusLevel

LOG4J_DEFAULT_STATUS_LOGGER

ERROR

The StatusLogger logs events that occur in the logging system to the console. During configuration, AbstractConfiguration registers a StatusConsoleListener with the StatusLogger that may redirect status log events from the default console output to a file. The listener also supports fine-grained filtering. This system property specifies the default status log level for the listener to use if the configuration does not specify a status level.

Note: this property is used by the log4j-core implementation only after a configuration file has been found.
The initial "listenersLevel" of the StatusLogger. If StatusLogger listeners are added, the "listenerLevel" is changed to that of the most verbose listener. If any listeners are registered, the listenerLevel is used to quickly determine if an interested listener exists.

By default, StatusLogger listeners are added when a configuration is found and by the JMX StatusLoggerAdmin MBean. For example, if a configuration contains `<Configuration status="trace">`, a listener with statusLevel TRACE is registered and the StatusLogger listenerLevel is set to TRACE, resulting in verbose status messages displayed on the console.

If no listeners are registered, the listenersLevel is not used, and the StatusLogger output level is determined by StatusLogger.getLogger().getLevel() (see property org.apache.logging.log4j.simplelog.StatusLogger.level).

Number of StatusLogger events that are kept in a buffer and can be retrieved with StatusLogger.getStatusData().
<table>
<thead>
<tr>
<th>Configuration Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log4j2.statusLoggerDateformat</code></td>
<td>Date-time format string to use as the format for timestamps in the status logger output. See java.text.SimpleDateFormat for supported formats.</td>
</tr>
<tr>
<td><code>log4j2.asyncLoggerExceptionHandler</code></td>
<td>See Async Logger System Properties for details.</td>
</tr>
<tr>
<td><code>log4j2.asyncLoggerRingBufferSize</code></td>
<td>See Async Logger System Properties for details.</td>
</tr>
<tr>
<td><code>log4j2.asyncLoggerWaitStrategy</code></td>
<td>See Async Logger System Properties for details.</td>
</tr>
<tr>
<td><code>log4j2.asyncLoggerConfigExceptionHandler</code></td>
<td>See Mixed Async/Synchronous Logger System Properties for details.</td>
</tr>
<tr>
<td><code>log4j2.asyncLoggerConfigRingBufferSize</code></td>
<td>See Mixed Async/Synchronous Logger System Properties for details.</td>
</tr>
<tr>
<td><code>log4j2.julLoggerAdapter</code></td>
<td>Default LoggerFactory to use in the JUL adapter.</td>
</tr>
<tr>
<td><code>log4j2.formatMsgAsync</code></td>
<td>If false (the default), Log4j will make sure the message is formatted in the caller thread, to ensure the value at the time of the call to the logger is the value that is logged.</td>
</tr>
</tbody>
</table>
### Configuration

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log4j2.asyncQueueFullPolicy</td>
<td>Used by Async Loggers and the AsyncAppender to maintain application throughput even when the underlying appender cannot keep up with the logging rate and the queue is filling up. If no value is specified (the default) events are never discarded. If the queue is full, the logger call blocks until the event can be added to the queue. Specify Discard to drop events whose level is equal or less than the threshold level (INFO by default) when the queue is full.</td>
</tr>
<tr>
<td>log4j2.discardThreshold</td>
<td>Used by the DiscardingAsyncQueueFullPolicy to determine which events to drop when the queue becomes full. By default, INFO, DEBUG and TRACE level events are discarded when the queue is full. This property only has effect if Discard is specified as the log4j2.AsyncQueueFullPolicy.</td>
</tr>
<tr>
<td>log4j2.messageFactory</td>
<td>Default message factory used by Loggers if no factory was specified.</td>
</tr>
<tr>
<td>log4j2.flowMessageFactory</td>
<td>Default flow message factory used by Loggers.</td>
</tr>
<tr>
<td>log4j2.isWebapp</td>
<td>This system property can be used to force Log4j 2 to behave as if it is part of a web application (when true) or as if it is not part of a web application (when false).</td>
</tr>
</tbody>
</table>
log4j2.enableThreadlocals LOG4J_ENABLE_THREAD true (log4j2.enable.threadlocals)

This system property can be used to switch off the use of threadlocals, which will partly disable Log4j's garbage-free behaviour: to be fully garbage-free, Log4j stores objects in ThreadLocal fields to reuse them, otherwise new objects are created for each log event. Note that this property is not effective when Log4j detects it is running in a web application.

log4j2.enableDirectEncoder LOG4J_ENABLE_DIRECT_TRUE true (log4j2.enable.direct.encode)

This property can be used to force garbage-aware Layouts and Appenders to revert to the pre-2.6 behaviour where converting log events to text generates temporary objects like Strings and char[] arrays, and converting this text to bytes generates temporary byte[] arrays. By default, this property is true and garbage-aware Layouts and Appenders that convert log events to text will convert this text to bytes without creating temporary objects.

log4j2.initialReusableMsgSize LOG4J_INITIAL_REUSABLE 128 (log4j.initialReusableMsgSize)

In GC-free mode, this property determines the initial size of the reusable StringBuilders where the message text is formatted and potentially passed to background threads.

log4j2.maxReusableMsgSize LOG4J_MAX_REUSABLE_518 (log4j.maxReusableMsgSize)

In GC-free mode, this property determines the maximum size of the reusable StringBuilders where the message text is formatted and potentially passed to background threads.

log4j2.layoutStringBuilderMaxSize LOG4J_LAYOUT_STRING_BUILDER_MAX_SIZE 2048 (log4j.layoutStringBuilder.maxSize)

This property determines the maximum size of the thread-local reusable StringBuilders used to format the log event to text by Layouts that extend AbstractStringLayout.
<table>
<thead>
<tr>
<th>Configuration Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log4j2.unboxRingbufferSize</td>
<td>The org.apache.logging.log4j.util.Unbox utility manages a small thread-local ring buffer of StringBuilders. Each time one of the box() methods is called, the next slot in the ring buffer is used, until the ring buffer is full and the first slot is reused. By default the Unbox ring buffer has 32 slots, so user code can have up to 32 boxed primitives in a single logger call. If more slots are required, set system property log4j.unbox.ringbuffer.size to the desired ring buffer size. Note that the specified number will be rounded up to the nearest power of 2.</td>
</tr>
<tr>
<td>log4j2.loggerContextStacktraceOnStart</td>
<td>Prints a stacktrace to the status logger at DEBUG level when the LoggerContext is started. For debug purposes.</td>
</tr>
<tr>
<td>log4j2.formatMsgNoLookup</td>
<td>Disables message pattern lookups globally when set to true. This is equivalent to defining all message patterns using %m{nolookups}.</td>
</tr>
</tbody>
</table>

Log4j 2 global configuration properties
6 Web Applications and JSPs

6.1 Using Log4j 2 in Web Applications

You must take particular care when using Log4j or any other logging framework within a Java EE web application. It's important for logging resources to be properly cleaned up (database connections closed, files closed, etc.) when the container shuts down or the web application is undeployed. Because of the nature of class loaders within web applications, Log4j resources cannot be cleaned up through normal means. Log4j must be "started" when the web application deploys and "shut down" when the web application undeploys. How this works varies depending on whether your application is a Servlet 3.0 or newer or Servlet 2.5 web application.

In either case, you'll need to add the log4j-web module to your deployment as detailed in the Maven, Ivy, and Gradle Artifacts manual page.

To avoid problems the Log4j shutdown hook will automatically be disabled when the log4j-web jar is included.

6.1.1 Configuration

Log4j allows the configuration file to be specified in web.xml using the log4jConfiguration context parameter. Log4j will search for configuration files by:

1. If a location is provided it will be searched for as a servlet context resource. For example, if log4jConfiguration contains "logging.xml" then Log4j will look for a file with that name in the root directory of the web application.
2. If no location is defined Log4j will search for a file that starts with "log4j2" in the WEB-INF directory. If more than one file is found, and if a file that starts with "log4j2-name" is present, where name is the name of the web application, then it will be used. Otherwise the first file will be used.
3. The "normal" search sequence using the classpath and file URLs will be used to locate the configuration file.

6.1.2 Servlet 3.0 and Newer Web Applications

A Servlet 3.0 or newer web application is any <web-app> whose version attribute has a value of "3.0" or higher. Of course, the application must also be running in a compatible web container. Some examples are: Tomcat 7.0 and higher, GlassFish 3.0 and higher, JBoss 7.0 and higher, Oracle WebLogic 12c and higher, and IBM WebSphere 8.0 and higher.

6.1.2.1 The Short Story

Log4j 2 "just works" in Servlet 3.0 and newer web applications. It is capable of automatically starting when the application deploys and shutting down when the application undeploys.

Thanks to the ServletContainerInitializer API added to Servlet 3.0, the relevant Filter and ServletContextListener classes can be registered dynamically on web application startup.

Important Note! For performance reasons, containers often ignore certain JARs known not to contain TLDs or ServletContainerInitializers and do not scan them for web-fragments and initializers. Importantly, Tomcat 7 <7.0.43 ignores all JAR files named log4j*.jar, which prevents this feature from working. This has been fixed in Tomcat 7.0.43, Tomcat 8, and later. In Tomcat 7 <7.0.43 you will need to change catalina.properties and remove "log4j*.jar" from the jarsToSkip
property. You may need to do something similar on other containers if they skip scanning Log4j JAR files.

6.1.2.2 The Long Story
The Log4j 2 Web JAR file is a web-fragment configured to order before any other web fragments in your application. It contains a ServletContainerInitializer (Log4jServletContainerInitializer) that the container automatically discovers and initializes. This adds the Log4jServletContextListener and Log4jServletFilter to the ServletContext. These classes properly initialize and deinitialize the Log4j configuration.

For some users, automatically starting Log4j is problematic or undesirable. You can easily disable this feature using the isLog4jAutoInitializationDisabled context parameter. Simply add it to your deployment descriptor with the value "true" to disable auto-initialization. You must define the context parameter in web.xml. If you set it programmatically, it will be too late for Log4j to detect the setting.

```xml
<context-param>
  <param-name>isLog4jAutoInitializationDisabled</param-name>
  <param-value>true</param-value>
</context-param>
```

Once you disable auto-initialization, you must initialize Log4j as you would a Servlet 2.5 web application. You must do so in a way that this initialization happens before any other application code (such as Spring Framework startup code) executes.

You can customize the behavior of the listener and filter using the log4jContextName, log4jConfiguration, and/or isLog4jContextSelectorNamed context parameters. Read more about this in the Context Parameters section below. You must not manually configure the Log4jServletContextListener or Log4jServletFilter in your deployment descriptor (web.xml) or in another initializer or listener in a Servlet 3.0 or newer application unless you disable auto-initialization with isLog4jAutoInitializationDisabled. Doing so will result in startup errors and unspecified erroneous behavior.

6.1.3 Servlet 2.5 Web Applications

A Servlet 2.5 web application is any <web-app> whose version attribute has a value of "2.5." The version attribute is the only thing that matters; even if the web application is running in a Servlet 3.0 or newer container, it is a Servlet 2.5 web application if the version attribute is "2.5." Note that Log4j 2 does not support Servlet 2.4 and older web applications.

If you are using Log4j in a Servlet 2.5 web application, or if you have disabled auto-initialization with the isLog4jAutoInitializationDisabled context parameter, you must configure the Log4jServletContextListener and Log4jServletFilter in the deployment descriptor or programmatically. The filter should match all requests of any type. The listener should be the very first listener defined in your application, and the filter should be the very first filter defined and mapped in your application. This is easily accomplished using the following web.xml code:
<listener>
   <listener-class>org.apache.logging.log4j.web.Log4jServletContextListener</listener-class>
</listener>

<filter>
   <filter-name>log4jServletFilter</filter-name>
   <filter-class>org.apache.logging.log4j.web.Log4jServletFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>log4jServletFilter</filter-name>
   <url-pattern>/*</url-pattern>
   <dispatcher>REQUEST</dispatcher>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>INCLUDE</dispatcher>
   <dispatcher>ERROR</dispatcher>
   <dispatcher>ASYNC</dispatcher><!-- Servlet 3.0 w/ disabled auto-initialization only; not supported in
   </filter-mapping>

You can customize the behavior of the listener and filter using the log4jContextName, log4jConfiguration, and/or isLog4jContextSelectorNamed context parameters. Read more about this in the Context Parameters section below.

6.1.4 Context Parameters

By default, Log4j 2 uses the ServletContext's context name as the LoggerContext name and uses the standard pattern for locating the Log4j configuration file. There are three context parameters that you can use to control this behavior. The first, isLog4jContextSelectorNamed, specifies whether the context should be selected using the JndiContextSelector. If isLog4jContextSelectorNamed is not specified or is anything other than true, it is assumed to be false.

If isLog4jContextSelectorNamed is true, log4jContextName must be specified or display-name must be specified in web.xml; otherwise, the application will fail to start with an exception. log4jConfiguration should also be specified in this case, and must be a valid URI for the configuration file; however, this parameter is not required.

If isLog4jContextSelectorNamed is not true, log4jConfiguration may optionally be specified and must be a valid URI or path to a configuration file or start with "classpath:" to denote a configuration file that can be found on the classpath. Without this parameter, Log4j will use the standard mechanisms for locating the configuration file.

When specifying these context parameters, you must specify them in the deployment descriptor (web.xml) even in a Servlet 3.0 or never application. If you add them to the ServletContext within a listener, Log4j will initialize before the context parameters are available and they will have no effect. Here are some sample uses of these context parameters.

6.1.4.1 Set the Logging Context Name to "myApplication"

<context-param>
   <param-name>log4jContextName</param-name>
   <param-value>myApplication</param-value>
</context-param>
6.1.4.2 Set the Configuration Path/File/URI to "/etc/myApp/myLogging.xml"

```xml
<context-param>
    <param-name>log4jConfiguration</param-name>
    <param-value>file:///etc/myApp/myLogging.xml</param-value>
</context-param>
```

6.1.4.3 Use the JndiContextSelector

```xml
<context-param>
    <param-name>isLog4jContextSelectorNamed</param-name>
    <param-value>true</param-value>
</context-param>
<context-param>
    <param-name>log4jContextName</param-name>
    <param-value>appWithJndiSelector</param-value>
</context-param>
<context-param>
    <param-name>log4jConfiguration</param-name>
    <param-value>file:///D:/conf/myLogging.xml</param-value>
</context-param>

Note that in this case you must also set the "Log4jContextSelector" system property to "org.apache.logging.log4j.core.selector.JndiContextSelector".

6.1.5 Using Web Application Information During the Configuration

You may want to use information about the web application during configuration. For example, you could embed the web application’s context path in the name of a Rolling File Appender. See WebLookup in Lookups for more information.

6.1.6 JavaServer Pages Logging

You may use Log4j 2 within JSPs just as you would within any other Java code. Simple obtain a Logger and call its methods to log events. However, this requires you to use Java code within your JSPs, and some development teams rightly are not comfortable with doing this. If you have a dedicated user interface development team that is not familiar with using Java, you may even have Java code disabled in your JSPs.

For this reason, Log4j 2 provides a JSP Tag Library that enables you to log events without using any Java code. To read more about using this tag library, read the Log4j Tag Library documentation.

Important Note! As noted above, containers often ignore certain JARs known not to contain TLDs and do not scan them for TLD files. Importantly, Tomcat 7 <7.0.43 ignores all JAR files named log4j*.jar, which prevents the JSP tag library from being automatically discovered. This does not affect Tomcat 6.x and has been fixed in Tomcat 7.0.43, Tomcat 8, and later. In Tomcat 7 <7.0.43 you will need to change catalina.properties and remove "log4j*.jar" from the jarsToSkip property. You may need to do something similar on other containers if they skip scanning Log4j JAR files.

6.1.7 Asynchronous Requests and Threads
The handling of asynchronous requests is tricky, and regardless of Servlet container version or configuration Log4j cannot handle everything automatically. When standard requests, forwards, includes, and error resources are processed, the Log4jServletFilter binds the LoggerContext to the thread handling the request. After request processing completes, the filter unbinds the LoggerContext from the thread.

Similarly, when an internal request is dispatched using a javax.servlet.AsyncContext, the Log4jServletFilter also binds the LoggerContext to the thread handling the request and unbinds it when request processing completes. However, this only happens for requests dispatched through the AsyncContext. There are other asynchronous activities that can take place other than internal dispatched requests.

For example, after starting an AsyncContext you could start up a separate thread to process the request in the background, possibly writing the response with the ServletOutputStream. Filters cannot intercept the execution of this thread. Filters also cannot intercept threads that you start in the background during non-asynchronous requests. This is true whether you use a brand new thread or a thread borrowed from a thread pool. So what can you do for these special threads?

You may not need to do anything. If you didn’t use the isLog4jContextSelectorNamed context parameter, there is no need to bind the LoggerContext to the thread. Log4j can safely locate the LoggerContext on its own. In these cases, the filter provides only very modest performance gains, and only when creating new Loggers. However, if you did specify the isLog4jContextSelectorNamed context parameter with the value "true", you will need to manually bind the LoggerContext to asynchronous threads. Otherwise, Log4j will not be able to locate it.

Thankfully, Log4j provides a simple mechanism for binding the LoggerContext to asynchronous threads in these special circumstances. The simplest way to do this is to wrap the Runnable instance that is passed to the AsyncContext.start() method.
import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.web.WebLoggerContextUtils;

public class TestAsyncServlet extends HttpServlet {

    @Override
    protected void doGet(final HttpServletRequest req, final HttpServletResponse resp) throws ServletException, IOException {
        final AsyncContext asyncContext = req.startAsync();
        asyncContext.start(WebLoggerContextUtils.wrapExecutionContext(this.getServletContext(), new Runnable() {
            @Override
            public void run() {
                final Logger logger = LogManager.getLogger(TestAsyncServlet.class);
                logger.info("Hello, servlet!");
            }
        }));
    }

    @Override
    protected void doPost(final HttpServletRequest req, final HttpServletResponse resp) throws ServletException, IOException {
        final AsyncContext asyncContext = req.startAsync();
        asyncContext.start(new Runnable() {
            @Override
            public void run() {
                final Log4jWebSupport webSupport = WebLoggerContextUtils.getWebLifeCycle(TestAsyncServlet.this.getServletContext());
                webSupport.setLoggerContext();
                // do stuff
                webSupport.clearLoggerContext();
            }
        }));
    }
}

This can be slightly more convenient when using Java 1.8 and lambda functions as demonstrated below.
import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.web.WebLoggerContextUtils;

public class TestAsyncServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        final AsyncContext asyncContext = req.startAsync();
        asyncContext.start(WebLoggerContextUtils.wrapExecutionContext(this.getServletContext(), () -> {
            final Logger logger = LogManager.getLogger(TestAsyncServlet.class);
            logger.info("Hello, servlet!");
        }));
    }
}

Alternatively, you can obtain the Log4jWebLifeCycle instance from the ServletContext attributes, call its setLoggerContext method as the very first line of code in your asynchronous thread, and call its clearLoggerContext method as the very last line of code in your asynchronous thread. The following code demonstrates this. It uses the container thread pool to execute asynchronous request processing, passing an anonymous inner Runnable to the start method.
import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.web.Log4jWebLifeCycle;
import org.apache.logging.log4j.web.WebLoggerContextUtils;

public class TestAsyncServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        final AsyncContext asyncContext = req.startAsync();
        asyncContext.start(new Runnable() {
            @Override
            public void run() {
                final Log4jWebLifeCycle webLifeCycle = WebLoggerContextUtils.getWebLifeCycle(TestAsyncServlet.this.getServletContext());
                webLifeCycle.setLoggerContext();
                try {
                    final Logger logger = LogManager.getLogger(TestAsyncServlet.class);
                    logger.info("Hello, servlet!");
                } finally {
                    webLifeCycle.clearLoggerContext();
                }
            }
        });
    }
}

Note that you must call clearLoggerContext once your thread is finished processing. Failing
to do so will result in memory leaks. If using a thread pool, it can even disrupt the logging of other
web applications in your container. For that reason, the example here shows clearing the context in a
finally block, which will always execute.

6.1.8 Using the Servlet Appender

Log4j provides a Servlet Appender that uses the servlet context as the log target. For example:
To avoid double logging of exceptions to the servlet context, you must use `@ex{none}` in your `PatternLayout` as shown in the example. The exception will be omitted from the message text but it is passed to the servlet context as the actual Throwable object.
7 Plugins

7.1 Plugins

7.1.1 Introduction

Log4j 1.x allowed for extension by requiring class attributes on most of the configuration declarations. In the case of some elements, notably the PatternLayout, the only way to add new pattern converters was to extend the PatternLayout class and add them via code. One goal of Log4j 2 is to make extending it extremely easy through the use of plugins.

In Log4j 2 a plugin is declared by adding a `@Plugin` annotation to the class declaration. During initialization the `Configuration` will invoke the `PluginManager` to load the built-in Log4j plugins as well as any custom plugins. The `PluginManager` locates plugins by looking in five places:

1. Serialized plugin listing files on the classpath. These files are generated automatically during the build (more details below).
2. (OSGi only) Serialized plugin listing files in each active OSGi bundle. A `BundleListener` is added on activation to continue checking new bundles after `log4j-core` has started.
3. A comma-separated list of packages specified by the `log4j.plugin.packages` system property.
4. Packages passed to the static `PluginManager.addPackages` method (before Log4j configuration occurs).
5. The `packages` declared in your `log4j2` configuration file.

If multiple Plugins specify the same (case-insensitive) `name`, then the load order above determines which one will be used. For example, to override the `File` plugin which is provided by the built-in `FileAppender` class, you would need to place your plugin in a JAR file in the CLASSPATH ahead of `log4j-core.jar`. This is not recommended; plugin name collisions will cause a warning to be emitted. Note that in an OSGi environment, the order that bundles are scanned for plugins generally follows the same order that bundles were installed into the framework. See `getBundles()` and `SynchronousBundleListener`. In short, name collisions are even more unpredictable in an OSGi environment.

Serialized plugin listing files are generated by an annotation processor contained in the `log4j-core` artifact which will automatically scan your code for Log4j 2 plugins and output a metadata file in your processed classes. There is nothing extra that needs to be done to enable this; the Java compiler will automatically pick up the annotation processor on the class path unless you explicitly disable it. In that case, it would be important to add another compiler pass to your build process that only handles annotation processing using the `Log4j 2` annotation processor class, `org.apache.logging.log4j.core.config.plugins.processor.PluginProcessor`. To do this using Apache Maven, add the following execution to your `maven-compiler-plugin` (version 2.2 or higher) build plugin:
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <executions>
    <execution>
      <id>log4j-plugin-processor</id>
      <goals>
        <goal>compile</goal>
      </goals>
      <phase>process-classes</phase>
      <configuration>
        <proc>only</proc>
        <annotationProcessors>
          <annotationProcessor>org.apache.logging.log4j.core.config.plugins.processor.PluginProcessor</annotationProcessor>
        </annotationProcessors>
      </configuration>
    </execution>
  </executions>
</plugin>

As the configuration is processed the appropriate plugins will be automatically configured and initialized. Log4j 2 utilizes a few different categories of plugins which are described in the following sections.

7.1.2 Core

Core plugins are those that are directly represented by an element in a configuration file, such as an Appender, Layout, Logger or Filter. Custom plugins that conform to the rules laid out in the next paragraph may simply be referenced in the configuration, provided they are appropriate configured to be loaded by the PluginManager.

Every Core plugin must declare a static method annotated with @PluginFactory or @PluginBuilderFactory. The former is used for static factory methods that provide all options as method parameters, and the latter is used to construct a new Builder<T> class whose fields are used for injecting attributes and child nodes. To allow the Configuration to pass the correct parameters to the method, every parameter to the method must be annotated as one of the following attribute types. Each attribute or element annotation must include the name that must be present in the configuration in order to match the configuration item to its respective parameter. For plugin builders, the names of the fields will be used by default if no name is specified in the annotation. There are dozens of plugins in Log4j Core that can be used as examples for more complex scenarios including hierarchical builder classes (e.g., see FileAppender). See Extending Log4j with Plugin Builders for more details.

7.1.2.1 Attribute Types

PluginAttribute

The parameter must be convertible from a String using a TypeConverter. Most built-in types are already supported, but custom TypeConverter plugins may also be provided for more type support. Note that PluginBuilderAttribute can be used in builder class fields as an easier way to provide default values.
PluginElement
The parameter may represent a complex object that itself has parameters that can be configured. This also supports injecting an array of elements.

PluginConfiguration
The current Configuration object will be passed to the plugin as a parameter.

PluginNode
The current Node being parsed will be passed to the plugin as a parameter.

PluginValue
The value of the current Node or its attribute named value.

7.1.2.2 Constraint Validators
Plugin factory fields and parameters can be automatically validated at runtime using constraint validators inspired by the Bean Validation spec. The following annotations are bundled in Log4j, but custom ConstraintValidators can be created as well.

Required
This annotation validates that a value is non-empty. This covers a check for null as well as several other scenarios: empty CharSequence objects, empty arrays, empty Collection instances, and empty Map instances.

ValidHost
This annotation validates that a value corresponds to a valid hostname. This uses the same validation as InetAddress::getByName.

ValidPort
This annotation validates that a value corresponds to a valid port number between 0 and 65535.

7.1.3 Converters
Converters are used by PatternLayout to render the elements identified by the conversion pattern. Every converter must specify its category as "Converter" on the @Plugin annotation, have a static newInstance method that accepts an array of Strings as its only parameter and returns an instance of the Converter, and must have a @ConverterKeys annotation present that contains the array of converter patterns that will cause the Converter to be selected. Converters that are meant to handle LogEvents must extend the LogEventPatternConverter class and must implement a format method that accepts a LogEvent and a StringBuilder as arguments. The Converter should append the result of its operation to the StringBuilder.

A second type of Converter is the FileConverter - which must have "FileConverter" specified in the category attribute of the @Plugin annotation. While similar to a LogEventPatternConverter, instead of a single format method these Converters will have two variations; one that takes an Object and one that takes an array of Objects instead of the LogEvent. Both append to the provided StringBuilder in the same fashion as a LogEventPatternConverter. These Converters are typically used by the RollingFileAppender to construct the name of the file to log to.

If multiple Converters specify the same ConverterKeys, then the load order above determines which one will be used. For example, to override the %date converter which is provided by the built-in DatePatternConverter class, you would need to place your plugin in a JAR file in
the CLASSPATH ahead of log4j-core.jar. This is not recommended; pattern ConverterKeys collisions will cause a warning to be emitted. Try to use unique ConverterKeys for your custom pattern converters.

### 7.1.4 KeyProviders

Some components within Log4j may provide the ability to perform data encryption. These components require a secret key to perform the encryption. Applications may provide the key by creating a class that implements the `SecretKeyProvider` interface.

### 7.1.5 Lookups

Lookups are perhaps the simplest plugins of all. They must declare their type as "Lookup" on the plugin annotation and must implement the `StrLookup` interface. They will have two methods; a lookup method that accepts a String key and returns a String value and a second lookup method that accepts both a LogEvent and a String key and returns a String. Lookups may be referenced by specifying `${name:key}` where name is the name specified in the Plugin annotation and key is the name of the item to locate.

### 7.1.6 TypeConverters

`TypeConverters` are a sort of meta-plugin used for converting strings into other types in a plugin factory method parameter. Other plugins can already be injected via the `@PluginElement` annotation; now, any type supported by the type conversion system can be used in a `@PluginAttribute` parameter. Conversion of enum types are supported on demand and do not require custom `TypeConverter` classes. A large number of built-in Java classes are already supported; see `TypeConverters` for a more exhaustive listing.

Unlike other plugins, the plugin name of a `TypeConverter` is purely cosmetic. Appropriate type converters are looked up via the `Type` interface rather than via `Class<?>` objects only. Do note that `TypeConverter` plugins must have a default constructor.

### 7.2 Developer Notes

If a plugin class implements `Collection` or `Map`, then no factory method is used. Instead, the class is instantiated using the default constructor, and all child configuration nodes are added to the `Collection` or `Map`. 
8 Lookups

8.1 Lookups
Lookups provide a way to add values to the Log4j configuration at arbitrary places. They are a particular type of Plugin that implements the StrLookup interface. Information on how to use Lookups in configuration files can be found in the Property Substitution section of the Configuration page.

8.1.1 Context Map Lookup
The ContextMapLookup allows applications to store data in the Log4j ThreadContext Map and then retrieve the values in the Log4j configuration. In the example below, the application would store the current user's login id in the ThreadContext Map with the key "loginId". During initial configuration processing the first '$' will be removed. The PatternLayout supports interpolation with Lookups and will then resolve the variable for each event. Note that the pattern "%X{loginId}" would achieve the same result.

```xml
<File name="Application" fileName="application.log">
  <PatternLayout>
    <pattern>%d %p %c{1.} [%t] $${ctx:loginId} %m%n</pattern>
  </PatternLayout>
</File>
```

8.1.2 Date Lookup
The DateLookup is somewhat unusual from the other lookups as it doesn't use the key to locate an item. Instead, the key can be used to specify a date format string that is valid for SimpleDateFormat. The current date, or the date associated with the current log event will be formatted as specified.

```xml
<RollingFile name="Rolling-${map:type}" fileName="${filename}" filePattern="target/rolling1/test1-${date:MM-dd-yyyy}.%i.log.gz">
  <PatternLayout>
    <pattern>%d %p %c{1.} [%t] %m%n</pattern>
  </PatternLayout>
  <SizeBasedTriggeringPolicy size="500" />
</RollingFile>
```

8.1.3 Environment Lookup
The EnvironmentLookup allows systems to configure environment variables, either in global files such as /etc/profile or in the startup scripts for applications, and then retrieve those variables from within the logging configuration. The example below includes the name of the currently logged in user in the application log.

```xml
<RollingFile name="Rolling-${map:type}" fileName="${filename}" filePattern="target/rolling1/test1-${date:MM-dd-yyyy}.%i.log.gz">
  <PatternLayout>
    <pattern>%d %p %c{1.} [%t] %m%n</pattern>
  </PatternLayout>
  <SizeBasedTriggeringPolicy size="500" />
</RollingFile>
```
This lookup also supports default value syntax. In the sample below, when the USER environment variable is undefined, the default value jdoe is used:

```xml
<File name="Application" fileName="application.log">
    <PatternLayout>
        <pattern>%d %p %c{1.} [%t] $$\{env:USER\} %m%n</pattern>
    </PatternLayout>
</File>
```

### 8.1.4 Java Lookup

The JavaLookup allows Java environment information to be retrieved in convenient preformatted strings using the `java:` prefix.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>The short Java version, like:</td>
</tr>
<tr>
<td>runtime</td>
<td>The Java runtime version, like:</td>
</tr>
<tr>
<td>vm</td>
<td>The Java VM version, like:</td>
</tr>
<tr>
<td>os</td>
<td>The OS version, like:</td>
</tr>
<tr>
<td>locale</td>
<td>Hardware information, like:</td>
</tr>
<tr>
<td>hw</td>
<td>Hardware information, like:</td>
</tr>
</tbody>
</table>

For example:
8.1.5 Jndi Lookup

The JndiLookup allows variables to be retrieved via JNDI. By default the key will be prefixed with java:comp/env/, however if the key contains a ":" no prefix will be added.

Java's JNDI module is not available on Android.

8.1.6 JVM Input Arguments Lookup (JMX)

Maps JVM input arguments -- but not main arguments -- using JMX to acquire the JVM arguments. Use the prefix jvmrunargs to access JVM arguments.

See the Javadocs for java.lang.management.RuntimeMXBean.getInputArguments().

Java's JMX module is not available on Android or on Google App Engine.

8.1.7 Log4j Configuration Location Lookup

Log4j configuration properties. The expressions ${log4j:configLocation} and ${log4j:configParentLocation} respectively provide the absolute path to the log4j configuration file and its parent folder.

The example below uses this lookup to place log files in a directory relative to the log4j configuration file.

8.1.8 Main Arguments Lookup (Application)

This lookup requires that you manually provide the main arguments of the application to Log4j:
import org.apache.logging.log4j.core.lookup.MainMapLookup;

public static void main(String args[]) {
    MainMapLookup.setMainArguments(args);
    ...
}

If the main arguments have been set, this lookup allows applications to retrieve these main argument values from within the logging configuration. The key that follows the main: prefix can either be a 0-based index into the argument list, or a string, where $\{\text{main:myString}\}$ is substituted with the value that follows myString in the main argument list.

For example, suppose the static void main String[] arguments are:

--file foo.txt --verbose -x bar

Then the following substitutions are possible:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>${main:0}</td>
<td>--file</td>
</tr>
<tr>
<td>${main:1}</td>
<td>foo.txt</td>
</tr>
<tr>
<td>${main:2}</td>
<td>--verbose</td>
</tr>
<tr>
<td>${main:3}</td>
<td>-x</td>
</tr>
<tr>
<td>${main:4}</td>
<td>bar</td>
</tr>
<tr>
<td>${main:-file}</td>
<td>foo.txt</td>
</tr>
<tr>
<td>${main:-x}</td>
<td>bar</td>
</tr>
<tr>
<td>${main:bar}</td>
<td>null</td>
</tr>
</tbody>
</table>

Example usage:

```xml
<File name="Application" fileName="application.log">
  <PatternLayout header="File: ${main:--file}">
    <Pattern>%d %m%n</Pattern>
  </PatternLayout>
</File>
```

8.1.9 Map Lookup

The MapLookup serves several purposes.

1. Provide the base for Properties declared in the configuration file.
2. Retrieve values from MapMessages in LogEvents.
3. Retrieve values set with MapLookup.setMainArguments(String[])

The first item simply means that the MapLookup is used to substitute properties that are defined in the configuration file. These variables are specified without a prefix - e.g. $\{\text{name}\}$. The second usage allows a value from the current MapMessage, if one is part of the current log event, to be substituted. In the example below the RoutingAppender will use a different RollingFileAppender for each unique value of the key named "type" in the MapMessage. Note that when used this way a value for "type" should be declared in the properties declaration to provide a default value in case the message is not a
MapMessage or the MapMessage does not contain the key. See the Property Substitution section of the Configuration page for information on how to set the default values.

```xml
<Routing name="Routing">
  <Routes pattern="$${map:type}"/>
  <Route>
    <RollingFile name="Rolling-$({map:type})" fileName="${filename}" filePattern="target/rolling1/test1-$({map:type}).%i.log.gz">
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <SizeBasedTriggeringPolicy size="500"/>
    </RollingFile>
  </Route>
</Routes>
</Routing>

8.1.10 Marker Lookup

The marker lookup allows you to use markers in interesting configurations like a routing appender. Consider the following YAML configuration and code that logs to different files based on markers:
Configuration:
status: debug

Appenders:
Console:
RandomAccessFile:
- name: SQL_APPENDER
  fileName: logs/sql.log
  PatternLayout:
    Pattern: "%d{ISO8601_BASIC} %-5level %logger{l} %X %msg%n"
- name: PAYLOAD_APPENDER
  fileName: logs/payload.log
  PatternLayout:
    Pattern: "%d{ISO8601_BASIC} %-5level %logger{l} %X %msg%n"
- name: PERFORMANCE_APPENDER
  fileName: logs/performance.log
  PatternLayout:
    Pattern: "%d{ISO8601_BASIC} %-5level %logger{l} %X %msg%n"

Routing:
name: ROUTING_APPENDER
Routes:
  pattern: "${marker:}
  Route:
    - key: PERFORMANCE
      ref: PERFORMANCE_APPENDER
    - key: PAYLOAD
      ref: PAYLOAD_APPENDER
    - key: SQL
      ref: SQL_APPENDER

Loggers:
Root:
  level: trace
  AppendRef:
    - ref: ROUTING_APPENDER

public static final Marker SQL = MarkerFactory.getMarker("SQL");
public static final Marker PAYLOAD = MarkerFactory.getMarker("PAYLOAD");
public static final Marker PERFORMANCE = MarkerFactory.getMarker("PERFORMANCE");

final Logger logger = LoggerFactory.getLogger(Logger.ROOT_LOGGER_NAME);

logger.info(SQL, "Message in Sql.log");
logger.info(PAYLOAD, "Message in Payload.log");
logger.info(PERFORMANCE, "Message in Performance.log");

Note the key part of the configuration is pattern: "${marker:}". This will produce three log files, each with a log event for a specific marker. Log4j will route the log event with the SQL marker to sql.log, the log event with the PAYLOAD marker to payload.log, and so on.

You can use the notation "$\{marker:name\}" and "$${marker:name}" to check for the existence of a marker where name is the marker name. If the marker exists, the expression returns the name, otherwise null.
8.1.11 Structured Data Lookup

The StructuredDataLookup is very similar to the MapLookup in that it will retrieve values from StructuredDataMessages. In addition to the Map values it will also return the name portion of the id (not including the enterprise number) and the type field. The main difference between the example below and the example for MapMessage is that the "type" is an attribute of the StructuredDataMessage while "type" would have to be an item in the Map in a MapMessage.

```xml
<Routing name="Routing">
  <Routes pattern="$${sd:type}">
    <Route>
      <RollingFile name="Rolling-${sd:type}" fileName="${filename}" filePattern="target/rolling1/test1-${sd:type}.%i.log.gz">
        <PatternLayout>
          <pattern>%d %p %c{1.} [%t] %m%n</pattern>
        </PatternLayout>
        <SizeBasedTriggeringPolicy size="500" />
      </RollingFile>
    </Route>
  </Routes>
</Routing>
```

8.1.12 System Properties Lookup

As it is quite common to define values inside and outside the application by using System Properties, it is only natural that they should be accessible via a Lookup. As system properties are often defined outside the application it would be quite common to see something like:

```xml
<Appenders>
  <File name="ApplicationLog" fileName="${sys:logPath}/app.log"/>
</Appenders>
```

This lookup also supports default value syntax. In the sample below, when the logPath system property is undefined, the default value /var/logs is used:

```xml
<Appenders>
  <File name="ApplicationLog" fileName="${sys:logPath:-/var/logs}/app.log"/>
</Appenders>
```

8.1.13 Web Lookup

The WebLookup allows applications to retrieve variables that are associated with the ServletContext. In addition to being able to retrieve various fields in the ServletContext, WebLookup supports looking up values stored as attributes or configured as initialization parameters. The following table lists various keys that can be retrieved:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
</table>

©2019, The Apache Software Foundation • ALL RIGHTS RESERVED.
attr. name
Returns the ServletContext attribute with the specified name

classContext
The class context of the application

effectiveMajorVersion
Gets the major version of the Servlet specification that the application represented by this ServletContext is based on.

effectiveMinorVersion
Gets the minor version of the Servlet specification that the application represented by this ServletContext is based on.

initParam. name
Returns the ServletContext initialization parameter with the specified name

majorVersion
Returns the major version of the Servlet API that this servlet container supports.

minorVersion
Returns the minor version of the Servlet API that this servlet container supports.

rootDir
Returns the result of calling getRealPath with a value of "/".

serverInfo
Returns the name and version of the servlet container on which the servlet is running.

servletContextName
Returns the name of the web application as defined in the display-name element of the deployment descriptor

Any other key names specified will first be checked to see if a ServletContext attribute exists with that name and then will be checked to see if an initialization parameter of that name exists. If the key is located then the corresponding value will be returned.

<Appenders>
    <File name="ApplicationLog" fileName="${web:rootDir}/app.log"/>
</Appenders>
9 Appenders

9.1 Appenders

Appenders are responsible for delivering LogEvents to their destination. Every Appender must implement the Appender interface. Most Appenders will extend AbstractAppender which adds Lifecycle and Filterable support. Lifecycle allows components to finish initialization after configuration has completed and to perform cleanup during shutdown. Filterable allows the component to have Filters attached to it which are evaluated during event processing.

Appenders usually are only responsible for writing the event data to the target destination. In most cases they delegate responsibility for formatting the event to a layout. Some appenders wrap other appenders so that they can modify the LogEvent, handle a failure in an Appender, route the event to a subordinate Appender based on advanced Filter criteria or provide similar functionality that does not directly format the event for viewing.

Appenders always have a name so that they can be referenced from Loggers.

In the tables below, the "Type" column corresponds to the Java type expected. For non-JDK classes, these should usually be in Log4j Core unless otherwise noted.

9.1.1 AsyncAppender

The AsyncAppender accepts references to other Appenders and causes LogEvents to be written to them on a separate Thread. Note that exceptions while writing to those Appenders will be hidden from the application. The AsyncAppender should be configured after the appenders it references to allow it to shut down properly.

By default, AsyncAppender uses java.util.concurrent.ArrayBlockingQueue which does not require any external libraries. Note that multi-threaded applications should exercise care when using this appender as such: the blocking queue is susceptible to lock contention and our tests showed performance may become worse when more threads are logging concurrently. Consider using lock-free Async Loggers for optimal performance.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppenderRef</td>
<td>String</td>
<td>The name of the Appenders to invoke asynchronously. Multiple AppenderRef elements can be configured.</td>
</tr>
<tr>
<td>blocking</td>
<td>boolean</td>
<td>If true, the appender will wait until there are free slots in the queue. If false, the event will be written to the error appender if the queue is full. The default is true.</td>
</tr>
<tr>
<td>shutdownTimeout</td>
<td>integer</td>
<td>How many milliseconds the Appender should wait to flush outstanding log events in the queue on shutdown. The default is zero which means to wait forever.</td>
</tr>
</tbody>
</table>
### AsyncAppender Parameters

There are also a few system properties that can be used to maintain application throughput even when the underlying appender cannot keep up with the logging rate and the queue is filling up. See the details for system properties `log4j2.AsyncQueueFullPolicy` and `log4j2.DiscardThreshold`.

A typical AsyncAppender configuration might look like:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufferSize</td>
<td>integer</td>
<td>Specifies the maximum number of events that can be queued. The default is 1024. Note that when using a disruptor-style BlockingQueue, this buffer size must be a power of 2. When the application is logging faster than the underlying appender can keep up with for a long enough time to fill up the queue, the behavior is determined by the AsyncQueueFullPolicy.</td>
</tr>
<tr>
<td>errorRef</td>
<td>String</td>
<td>The name of the Appender to invoke if none of the appenders can be called, either due to errors in the appenders or because the queue is full. If not specified then errors will be ignored.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is <code>true</code>, causing exceptions encountered while appending events to be internally logged and then ignored. When set to <code>false</code> exceptions will be propagated to the caller, instead. You must set this to <code>false</code> when wrapping this Appender in a FailoverAppender.</td>
</tr>
<tr>
<td>includeLocation</td>
<td>boolean</td>
<td>Extracting location is an expensive operation (it can make logging 5-20 times slower). To improve performance, location is not included by default when adding a log event to the queue. You can change this by setting <code>includeLocation=true</code>.</td>
</tr>
<tr>
<td>BlockingQueueFactory</td>
<td>BlockingQueueFactory</td>
<td>This element overrides what type of BlockingQueue to use. See below documentation for more details.</td>
</tr>
</tbody>
</table>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <File name="MyFile" fileName="logs/app.log">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
    </File>
    <Async name="Async">
      <AppenderRef ref="MyFile"/>
    </Async>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Async"/>
    </Root>
  </Loggers>
</Configuration>

Starting in Log4j 2.7, a custom implementation of BlockingQueue or TransferQueue can be specified using a BlockingQueueFactory plugin. To override the default BlockingQueueFactory, specify the plugin inside an <Async/> element like so:

<Configuration name="LinkedTransferQueueExample">
  <Appenders>
    <List name="List"/>
    <Async name="Async" bufferSize="262144">
      <AppenderRef ref="List"/>
      <LinkedTransferQueue/>
    </Async>
  </Appenders>
  <Loggers>
    <Root>
      <AppenderRef ref="Async"/>
    </Root>
  </Loggers>
</Configuration>

Log4j ships with the following implementations:

<table>
<thead>
<tr>
<th>Plugin Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayBlockingQueue</td>
<td>This is the default implementation that uses ArrayBlockingQueue.</td>
</tr>
<tr>
<td>DisruptorBlockingQueue</td>
<td>This uses the Conversant Disruptor implementation of BlockingQueue. This plugin takes a single optional attribute, spinPolicy, which corresponds to</td>
</tr>
<tr>
<td>JCToolsBlockingQueue</td>
<td>This uses JCTools, specifically the MPSC bounded lock-free queue.</td>
</tr>
</tbody>
</table>
LinkedTransferQueue

This uses the new in Java 7 implementation LinkedTransferQueue. Note that this queue does not use the bufferSize configuration attribute from AsyncAppender as LinkedTransferQueue does not support a maximum capacity.

BlockingQueueFactory Implementations

9.1.2 CassandraAppender

The CassandraAppender writes its output to an Apache Cassandra database. A keyspace and table must be configured ahead of time, and the columns of that table are mapped in a configuration file. Each column can specify either a StringLayout (e.g., a PatternLayout) along with an optional conversion type, or only a conversion type for org.apache.logging.log4j.spi.ThreadContextMap or org.apache.logging.log4j.spi.ThreadContextStack to store the MDC or NDC in a map or list column respectively. A conversion type compatible with java.util.Date will use the log event timestamp converted to that type (e.g., use java.util.Date to fill a timestamp column type in Cassandra).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>batched</td>
<td>boolean</td>
<td>Whether or not to use batch statements to write log messages to Cassandra. By default, this is false.</td>
</tr>
<tr>
<td>batchType</td>
<td>BatchStatement.Type</td>
<td>The batch type to use when using batched writes. By default, this is LOGGED.</td>
</tr>
<tr>
<td>bufferSize</td>
<td>int</td>
<td>The number of log messages to buffer or batch before writing. By default, no buffering is done.</td>
</tr>
<tr>
<td>clusterName</td>
<td>String</td>
<td>The name of the Cassandra cluster to connect to.</td>
</tr>
</tbody>
</table>
### columns

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColumnMapping[]</td>
<td>A list of column mapping configurations. Each column must specify a column name. Each column can have a conversion type specified by its fully qualified class name. By default, the conversion type is <code>String</code>. If the configured type is assignment-compatible with <code>ReadOnlyStringMap</code>, <code>ThreadContextMap</code>, or <code>ThreadContextStack</code>, then that column will be populated with the MDC or NDC respectively. If the configured type is assignment-compatible with <code>java.util.Date</code>, then the log timestamp will be converted to that configured date type. If a literal attribute is given, then its value will be used as is in the INSERT query without any escaping. Otherwise, the layout or pattern specified will be converted into the configured type and stored in that column.</td>
</tr>
</tbody>
</table>

### contactPoints

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SocketAddress[]</td>
<td>A list of hosts and ports of Cassandra nodes to connect to. These must be valid hostnames or IP addresses. By default, if a port is not specified for a host or it is set to 0, then the default Cassandra port of 9042 will be used. By default, <code>localhost:9042</code> will be used.</td>
</tr>
</tbody>
</table>

### filter

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
</tbody>
</table>

### ignoreExceptions

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>The default is <code>true</code>, causing exceptions encountered while appending events to be internally logged and then ignored. When set to <code>false</code> exceptions will be propagated to the caller, instead. You must set this to <code>false</code> when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>

### keyspace

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>The name of the keyspace containing the table that log messages will be written to.</td>
</tr>
</tbody>
</table>

### name

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
</tbody>
</table>

### password

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>The password to use (along with the username) to connect to Cassandra.</td>
</tr>
</tbody>
</table>
### CassandraAppender Parameters

Here is an example CassandraAppender configuration:

```xml
<Configuration name="CassandraAppenderTest">
  <Appenders>
    <Cassandra name="Cassandra" clusterName="Test Cluster" keyspace="test" table="logs" bufferSize="10" batched="true">
      <SocketAddress host="localhost" port="9042"/>
      <ColumnMapping name="id" pattern="%uuid{TIME}" type="java.util.UUID"/>
      <ColumnMapping name="timeid" literal="now()"/>
      <ColumnMapping name="message" pattern="%message"/>
      <ColumnMapping name="level" pattern="%level"/>
      <ColumnMapping name="marker" pattern="%marker"/>
      <ColumnMapping name="logger" pattern="%logger"/>
      <ColumnMapping name="timestamp" type="java.util.Date"/>
      <ColumnMapping name="mdc" type="org.apache.logging.log4j.spi.ThreadContextMap"/>
      <ColumnMapping name="ndc" type="org.apache.logging.log4j.spi.ThreadContextStack"/>
    </Cassandra>
  </Appenders>
  <Loggers>
    <Logger name="org.apache.logging.log4j.cassandra" level="DEBUG">
      <AppenderRef ref="Cassandra"/>
    </Logger>
  </Loggers>
</Configuration>
```

This example configuration uses the following table schema:
CREATE TABLE logs {
    id timeuuid PRIMARY KEY,
    timeid timeuuid,
    message text,
    level text,
    marker text,
    logger text,
    timestamp timestamp,
    mdc map<text,text>,
    ndc list<text>
};

9.1.3 ConsoleAppender
As one might expect, the ConsoleAppender writes its output to either System.out or System.err with System.out being the default target. A Layout must be provided to format the LogEvent.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of &quot;%m%n&quot; will be used.</td>
</tr>
<tr>
<td>follow</td>
<td>boolean</td>
<td>Identifies whether the appender honors reassignments of System.out or System.err via System.setOut or System.setErr made after configuration. Note that the follow attribute cannot be used with Jansi on Windows. Cannot be used with direct.</td>
</tr>
</tbody>
</table>
**ConsoleAppender Parameters**

A typical Console configuration might look like:

```xml
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

**9.1.4 FailoverAppender**

The FailoverAppender wraps a set of appenders. If the primary Appender fails the secondary appenders will be tried in order until one succeeds or there are no more secondaries to try.
## Parameter Name | Type  | Description |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>primary</td>
<td>String</td>
<td>The name of the primary Appender to use.</td>
</tr>
<tr>
<td>failovers</td>
<td>String[]</td>
<td>The names of the secondary Appenders to use.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>retryIntervalSeconds</td>
<td>integer</td>
<td>The number of seconds that should pass before retrying the primary Appender. The default is 60.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead.</td>
</tr>
<tr>
<td>target</td>
<td>String</td>
<td>Either &quot;SYSTEM_OUT&quot; or &quot;SYSTEM_ERR&quot;. The default is &quot;SYSTEM_ERR&quot;.</td>
</tr>
</tbody>
</table>

### FailoverAppender Parameters

A Failover configuration might look like:
9 Appenders

```xml
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz" ignoreExceptions="false">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
    <Console name="STDOUT" target="SYSTEM_OUT" ignoreExceptions="false">
      <PatternLayout pattern="%m%n"/>
    </Console>
    <Failover name="Failover" primary="RollingFile">
      <Failovers>
        <AppenderRef ref="Console"/>
      </Failovers>
    </Failover>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Failover"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.5 FileAppender

The FileAppender is an OutputStreamAppender that writes to the File named in the fileName parameter. The FileAppender uses a FileManager (which extends OutputStreamManager) to actually perform the file I/O. While FileAppenders from different Configurations cannot be shared, the FileManagers can be if the Manager is accessible. For example, two web applications in a servlet container can have their own configuration and safely write to the same file if Log4j is in a ClassLoader that is common to both of them.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>boolean</td>
<td>When true - the default, records will be appended to the end of the file. When set to false, the file will be cleared before new records are written.</td>
</tr>
<tr>
<td>bufferedIO</td>
<td>boolean</td>
<td>When true - the default, records will be written to a buffer and the data will be written to disk when the buffer is full or, if immediateFlush is set, when the record is written. File locking cannot be used with bufferedIO. Performance tests have shown that using buffered I/O significantly improves performance, even if immediateFlush is enabled.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bufferSize</td>
<td>int</td>
<td>When <code>bufferedIO</code> is true, this is the buffer size, the default is 8192 bytes.</td>
</tr>
<tr>
<td>createOnDemand</td>
<td>boolean</td>
<td>The appender creates the file on-demand. The appender only creates the file when a log event passes all filters and is routed to this appender. Defaults to false.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.</td>
</tr>
<tr>
<td>immediateFlush</td>
<td>boolean</td>
<td>When set to true - the default, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flushing after every write is only useful when using this appender with synchronous loggers. Asynchronous loggers and appenders will automatically flush at the end of a batch of events, even if immediateFlush is set to false. This also guarantees the data is written to disk but is more efficient.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of &quot;%m%n&quot; will be used.</td>
</tr>
<tr>
<td>locking</td>
<td>boolean</td>
<td>When set to true, I/O operations will occur only while the file lock is held allowing FileAppenders in multiple JVMs and potentially multiple hosts to write to the same file simultaneously. This will significantly impact performance so should be used carefully. Furthermore, on many systems the file lock is &quot;advisory&quot; meaning that other applications can perform operations on the file without acquiring a lock. The default value is false.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is <code>true</code>, causing exceptions encountered while appending events to be internally logged and then ignored. When set to <code>false</code> exceptions will be propagated to the caller, instead. You must set this to <code>false</code> when wrapping this Appender in a <code>FailoverAppender</code>.</td>
</tr>
<tr>
<td>filePermissions</td>
<td>String</td>
<td>File attribute permissions in POSIX format to apply whenever the file is created. Underlying files system shall support POSIX file attribute view. Examples: rw------- or rw-rw-rw- etc...</td>
</tr>
<tr>
<td>fileOwner</td>
<td>String</td>
<td>File owner to define whenever the file is created. Changing file’s owner may be restricted for security reason and Operation not permitted IOException thrown. Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file if <code>_POSIX_CHOWN_RESTRICTED</code> is in effect for path. Underlying files system shall support file owner attribute view.</td>
</tr>
<tr>
<td>fileGroup</td>
<td>String</td>
<td>File group to define whenever the file is created. Underlying files system shall support POSIX file attribute view.</td>
</tr>
</tbody>
</table>

### FileAppender Parameters

Here is a sample File configuration:
9.1.6 FlumeAppender

This is an optional component supplied in a separate jar.

Apache Flume is a distributed, reliable, and available system for efficiently collecting, aggregating, and moving large amounts of log data from many different sources to a centralized data store. The FlumeAppender takes LogEvents and sends them to a Flume agent as serialized Avro events for consumption.

The Flume Appender supports three modes of operation.

1. It can act as a remote Flume client which sends Flume events via Avro to a Flume Agent configured with an Avro Source.
2. It can act as an embedded Flume Agent where Flume events pass directly into Flume for processing.
3. It can persist events to a local BerkeleyDB data store and then asynchronously send the events to Flume, similar to the embedded Flume Agent but without most of the Flume dependencies.

Usage as an embedded agent will cause the messages to be directly passed to the Flume Channel and then control will be immediately returned to the application. All interaction with remote agents will occur asynchronously. Setting the "type" attribute to "Embedded" will force the use of the embedded agent. In addition, configuring agent properties in the appender configuration will also cause the embedded agent to be used.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agents</td>
<td>Agent[]</td>
<td>An array of Agents to which the logging events should be sent. If more than one agent is specified the first Agent will be the primary and subsequent Agents will be used in the order specified as secondaries should the primary Agent fail. Each Agent definition supplies the Agents host and port. The specification of agents and properties are mutually exclusive. If both are configured an error will result.</td>
</tr>
<tr>
<td>Property</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>agentRetries</td>
<td>integer</td>
<td>The number of times the agent should be retried before failing to a secondary. This parameter is ignored when type=&quot;persistent&quot; is specified (agents are tried once before failing to the next).</td>
</tr>
<tr>
<td>batchSize</td>
<td>integer</td>
<td>Specifies the number of events that should be sent as a batch. The default is 1. This parameter only applies to the Flume Appender.</td>
</tr>
<tr>
<td>compress</td>
<td>boolean</td>
<td>When set to true the message body will be compressed using gzip</td>
</tr>
<tr>
<td>connectTimeoutMillis</td>
<td>integer</td>
<td>The number of milliseconds Flume will wait before timing out the connection.</td>
</tr>
<tr>
<td>dataDir</td>
<td>String</td>
<td>Directory where the Flume write ahead log should be written. Valid only when embedded is set to true and Agent elements are used instead of Property elements.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>eventPrefix</td>
<td>String</td>
<td>The character string to prepend to each event attribute in order to distinguish it from MDC attributes. The default is an empty string.</td>
</tr>
<tr>
<td>flumeEventFactory</td>
<td>FlumeEventFactory</td>
<td>Factory that generates the Flume events from Log4j events. The default factory is the FlumeAvroAppender itself.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. If no layout is specified RFC5424Layout will be used.</td>
</tr>
<tr>
<td>lockTimeoutRetries</td>
<td>integer</td>
<td>The number of times to retry if a LockConflictException occurs while writing to Berkeley DB. The default is 5.</td>
</tr>
<tr>
<td>maxDelayMillis</td>
<td>integer</td>
<td>The maximum number of milliseconds to wait for batchSize events before publishing the batch.</td>
</tr>
<tr>
<td>mdcExcludes</td>
<td>String</td>
<td>A comma separated list of mdc keys that should be excluded from the FlumeEvent. This is mutually exclusive with the mdcIncludes attribute.</td>
</tr>
<tr>
<td>Property</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mdcIncludes</td>
<td>String</td>
<td>A comma separated list of mdc keys that should be included in the FlumeEvent. Any keys in the MDC not found in the list will be excluded. This option is mutually exclusive with the mdcExcludes attribute.</td>
</tr>
<tr>
<td>mdcRequired</td>
<td>String</td>
<td>A comma separated list of mdc keys that must be present in the MDC. If a key is not present a LoggingException will be thrown.</td>
</tr>
<tr>
<td>mdcPrefix</td>
<td>String</td>
<td>A string that should be prepended to each MDC key in order to distinguish it from event attributes. The default string is &quot;mdc:&quot;.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>properties</td>
<td>Property[]</td>
<td>One or more Property elements that are used to configure the Flume Agent. The properties must be configured without the agent name (the appender name is used for this) and no sources can be configured. Interceptors can be specified for the source using &quot;sources.log4j-source.interceptors&quot;. All other Flume configuration properties are allowed. Specifying both Agent and Property elements will result in an error. When used to configure in Persistent mode the valid properties are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. &quot;keyProvider&quot; to specify the name of the plugin to provide the secret key for encryption.</td>
</tr>
<tr>
<td>requestTimeoutMillis</td>
<td>integer</td>
<td>The number of milliseconds Flume will wait before timing out the request.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>
type | enumeration | One of "Avro", "Embedded", or "Persistent" to indicate which variation of the Appender is desired.
---|---|---

**FlumeAppender Parameters**

A sample FlumeAppender configuration that is configured with a primary and a secondary agent, compresses the body, and formats the body using the RFC5424Layout:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Flume name="eventLogger" compress="true">
      <Agent host="192.168.10.101" port="8800"/>
      <Agent host="192.168.10.102" port="8800"/>
      <RFC5424Layout enterpriseNumber="18060" includeMDC="true" appName="MyApp"/>
    </Flume>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="eventLogger"/>
    </Root>
  </Loggers>
</Configuration>
```

A sample FlumeAppender configuration that is configured with a primary and a secondary agent, compresses the body, formats the body using the RFC5424Layout, and persists encrypted events to disk:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Flume name="eventLogger" compress="true" type="persistent" dataDir="~/logData">
      <Agent host="192.168.10.101" port="8800"/>
      <Agent host="192.168.10.102" port="8800"/>
      <RFC5424Layout enterpriseNumber="18060" includeMDC="true" appName="MyApp"/>
      <Property name="keyProvider">MySecretProvider</Property>
    </Flume>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="eventLogger"/>
    </Root>
  </Loggers>
</Configuration>
```

A sample FlumeAppender configuration that is configured with a primary and a secondary agent, compresses the body, formats the body using RFC5424Layout and passes the events to an embedded Flume Agent.
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Flume name="eventLogger" compress="true" type="Embedded">
      <Agent host="192.168.10.101" port="8800"/>
      <Agent host="192.168.10.102" port="8800"/>
      <RFC5424Layout enterpriseNumber="18060" includeMDC="true" appName="MyApp"/>
    </Flume>
    <Console name="STDOUT">
      <PatternLayout pattern="%d [%p] %c %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="EventLogger" level="info">
      <AppenderRef ref="eventLogger"/>
    </Logger>
    <Root level="warn">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>

A sample FlumeAppender configuration that is configured with a primary and a secondary agent using Flume configuration properties, compresses the body, formats the body using RFC5424Layout and passes the events to an embedded Flume Agent.
```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error" name="MyApp" packages="">
  <Appenders>
    <Flume name="eventLogger" compress="true" type="Embedded">
      <Property name="channels">file</Property>
      <Property name="channels.file.type">file</Property>
      <Property name="channels.file.checkpointDir">target/file-channel/checkpoint</Property>
      <Property name="channels.file.dataDirs">target/file-channel/data</Property>
      <Property name="sinks">agent1 agent2</Property>
      <Property name="sinks.agent1.channel">file</Property>
      <Property name="sinks.agent1.type">avro</Property>
      <Property name="sinks.agent1.hostname">192.168.10.101</Property>
      <Property name="sinks.agent1.port">8800</Property>
      <Property name="sinks.agent1.batch-size">100</Property>
      <Property name="sinks.agent2.channel">file</Property>
      <Property name="sinks.agent2.type">avro</Property>
      <Property name="sinks.agent2.hostname">192.168.10.102</Property>
      <Property name="sinks.agent2.port">8800</Property>
      <Property name="sinks.agent2.batch-size">100</Property>
      <Property name="sinkgroups">group1</Property>
      <Property name="sinkgroups.group1.sinks">agent1 agent2</Property>
      <Property name="sinkgroups.group1.processor.type">failover</Property>
      <Property name="sinkgroups.group1.processor.priority.agent1">10</Property>
      <Property name="sinkgroups.group1.processor.priority.agent2">5</Property>
      <RFC5424Layout enterpriseNumber="18060" includeMDC="true" appName="MyApp"/>
    </Flume>
    <Console name="STDOUT">
      <PatternLayout pattern="%d [%p] %c %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="EventLogger" level="info">
      <AppenderRef ref="eventLogger"/>
    </Logger>
    <Root level="warn">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>

9.1.7 JDBCAppender

The JDBCAppender writes log events to a relational database table using standard JDBC. It can be configured to obtain JDBC connections using a JNDI DataSource or a custom factory method. Whichever approach you take, it must be backed by a connection pool. Otherwise, logging performance will suffer greatly. If batch statements are supported by the configured JDBC driver and a bufferSize is configured to be a positive number, then log events will be batched. Note that as of Log4j 2.8, there are two ways to configure log event to column mappings: the original ColumnConfig style that only allows strings and timestamps, and the new ColumnMapping plugin that uses Log4j’s built-in type conversion to allow for more data types (this is the same plugin as in the Cassandra Appender).
To get off the ground quickly during development, an alternative to using a connection source based on JNDI is to use the non-pooling `DriverManager` connection source. This connection source uses a JDBC connection string, a user name, and a password. Optionally, you can also use properties.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Required. The name of the Appender.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is <code>true</code>, causing exceptions encountered while appending events to be internally logged and then ignored. When set to <code>false</code> exceptions will be propagated to the caller, instead. You must set this to <code>false</code> when wrapping this Appender in a <code>FailoverAppender</code>.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>bufferSize</td>
<td>int</td>
<td>If an integer greater than 0, this causes the appender to buffer log events and flush whenever the buffer reaches this size.</td>
</tr>
<tr>
<td>connectionSource</td>
<td>ConnectionSource</td>
<td>Required. The connections source from which database connections should be retrieved.</td>
</tr>
<tr>
<td>tableName</td>
<td>String</td>
<td>Required. The name of the database table to insert log events into.</td>
</tr>
<tr>
<td>columnConfigs</td>
<td>ColumnConfig[]</td>
<td>Required (and/or columnMappings). Information about the columns that log event data should be inserted into and how to insert that data. This is represented with multiple <code>&lt;Column&gt;</code> elements.</td>
</tr>
</tbody>
</table>
### columnMappings

**Type:** ColumnMapping[]  
**Required (and/or columnConfigs):** A list of column mapping configurations. Each column must specify a column name. Each column can have a conversion type specified by its fully qualified class name. By default, the conversion type is String. If the configured type is assignment-compatible with `ReadOnlyStringMap` or `ThreadContextMap` or `ThreadContextStack`, then that column will be populated with the MDC or NDC respectively (this is database-specific how they handle inserting a `Map` or `List` value). If the configured type is assignment-compatible with `java.util.Date`, then the log timestamp will be converted to that configured date type. If the configured type is assignment-compatible with `java.sql.Clob` or `java.sql.NClob`, then the formatted event will be set as a Clob or NClob respectively (similar to the traditional ColumnConfig plugin).  
If a literal attribute is given, then its value will be used as is in the `INSERT` query without any escaping. Otherwise, the layout or pattern specified will be converted into the configured type and stored in that column.

<table>
<thead>
<tr>
<th>immediateFail</th>
<th>boolean</th>
<th>false</th>
</tr>
</thead>
</table>

When set to true, log events will not wait to try to reconnect and will fail immediately if the JDBC resources are not available. New in 2.11.2
reconnectIntervalMillis long 5000 If set to a value greater than 0, after an error, the JDBCDatabaseManager will attempt to reconnect to the database after waiting the specified number of milliseconds. If the reconnect fails then an exception will be thrown (which can be caught by the application if ignoreExceptions is set to false). New in 2.11.2.

JDBCAppender Parameters

When configuring the JDBCAppender, you must specify a ConnectionSource implementation from which the Appender gets JDBC connections. You must use exactly one of the following nested elements:

- `<DataSource>`: Uses JNDI.
- `<ConnectionFactory>`: Points to a class-method pair to provide JDBC connections.
- `<DriverManager>`: A quick and dirty way to get off the ground, no connection pooling.
- `<PoolingDriver>`: Uses Apache Commons DBCP to provide connection pooling.

### DataSource Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jndiName</td>
<td>String</td>
<td>Required. The full, prefixed JNDI name that the javax.sql.DataSource is bound to, such as java:/comp/env/jdbc/LoggingDatabase. The DataSource must be backed by a connection pool; otherwise, logging will be very slow.</td>
</tr>
</tbody>
</table>

### Class Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Class</td>
<td>Required. The fully qualified name of a class containing a static factory method for obtaining JDBC connections.</td>
</tr>
</tbody>
</table>
### ConnectionFactory Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionString</td>
<td>String</td>
<td>Required. The driver-specific JDBC connection string.</td>
</tr>
<tr>
<td>userName</td>
<td>String</td>
<td>The database user name. You cannot specify both properties and a user name or password.</td>
</tr>
<tr>
<td>password</td>
<td>String</td>
<td>The database password. You cannot specify both properties and a user name or password.</td>
</tr>
<tr>
<td>driverClassName</td>
<td>String</td>
<td>The JDBC driver class name. Some old JDBC Driver can only be discovered by explicitly loading them by class name.</td>
</tr>
<tr>
<td>properties</td>
<td>Property[]</td>
<td>A list of properties. You cannot specify both properties and a user name or password.</td>
</tr>
</tbody>
</table>

### DriverManager Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DriverManager parameters</td>
<td>DriverManager parameters</td>
<td>This connection source inherits all parameter from the DriverManager connection source.</td>
</tr>
</tbody>
</table>
### PoolingDriver Parameters (Apache Commons DBCP)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoCommitOnReturn</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>cacheState</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>ConnectionInitSqls</td>
<td>Strings</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>defaultAutoCommit</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>defaultCatalog</td>
<td>String</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>defaultQueryTimeoutSeconds</td>
<td>integer</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>defaultReadOnly</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>defaultTransactionIsolation</td>
<td>integer</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>disconnectionSqlCodes</td>
<td>Strings</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>fastFailValidation</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>maxConnLifetimeMillis</td>
<td>long</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>maxOpenPreparedStatements</td>
<td>integer</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>poolStatements</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>rollbackOnReturn</td>
<td>boolean</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>String</td>
<td>See Apache Commons DBCP PoolableConnectionFactory.</td>
</tr>
</tbody>
</table>
PoolableConnectionFactory Parameters (Apache Commons DBCP)

When configuring the JDBCAppender, use the nested `<Column>` elements to specify which columns in the table should be written to and how to write to them. The JDBCAppender uses this information to formulate a PreparedStatement to insert records without SQL injection vulnerability.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Required. The name of the database column.</td>
</tr>
<tr>
<td>pattern</td>
<td>String</td>
<td>Use this attribute to insert a value or values from the log event in this column using a PatternLayout pattern. Simply specify any legal pattern in this attribute. Either this attribute, literal, or isEventTimestamp=&quot;true&quot; must be specified, but not more than one of these.</td>
</tr>
<tr>
<td>literal</td>
<td>String</td>
<td>Use this attribute to insert a literal value in this column. The value will be included directly in the insert SQL, without any quoting (which means that if you want this to be a string, your value should contain single quotes around it like this: literal=&quot;'Literal String'&quot;). This is especially useful for databases that don't support identity columns. For example, if you are using Oracle you could specify literal=&quot;NAME_OF_YOUR_SEQUENCE.NEXTVAL&quot; to insert a unique ID in an ID column. Either this attribute, pattern, or isEventTimestamp=&quot;true&quot; must be specified, but not more than one of these.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| parameter      | String     | Use this attribute to insert an expression with a parameter marker `?` in this column. The value will be included directly in the insert SQL, without any quoting (which means that if you want this to be a string, your value should contain single quotes around it like this:  

```xml
<ColumnMapping  
name="instant"  
parameter="TIMESTAMPADD('MILLISECOND', ?, TIMESTAMP '1970-01-01')"/>
```

You can only specify one of literal or parameter.                                                                                               |
| isEventTimestamp | boolean   | Use this attribute to insert the event timestamp in this column, which should be a SQL datetime. The value will be inserted as a java.sql.Types.TIMESTAMP. Either this attribute (equal to true), pattern, or isEventTimestamp must be specified, but not more than one of these. |
| isUnicode      | boolean   | This attribute is ignored unless pattern is specified. If true or omitted (default), the value will be inserted as unicode (setNString or setNClob). Otherwise, the value will be inserted non-unicode (setString or setClob). |
| isClob         | boolean   | This attribute is ignored unless pattern is specified. Use this attribute to indicate that the column stores Character Large Objects (CLOBs). If true, the value will be inserted as a CLOB (setClob or setNClob). If false or omitted (default), the value will be inserted as a VARCHAR or NVARCHAR (setString or setNString). |
### ColumnMapping Parameters

Here are a couple sample configurations for the JDBCAppender, as well as a sample factory implementation that uses Commons Pooling and Commons DBCP to pool database connections:
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <JDBC name="databaseAppender" tableName="dbo.application_log">
      <DataSource jndiName="java:/comp/env/jdbc/LoggingDataSource" />
      <Column name="eventDate" isEventTimestamp="true" />
      <Column name="level" pattern="%level" />
      <Column name="logger" pattern="%logger" />
      <Column name="message" pattern="%message" />
      <Column name="exception" pattern="%ex{full}" />
    </JDBC>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <JDBC name="databaseAppender" tableName="LOGGING.APPLICATION_LOG">
      <ConnectionFactory class="net.example.db.ConnectionFactory" method="getDatabaseConnection" />
      <Column name="EVENT_ID" literal="LOGGING.APPLICATION_LOG_SEQUENCE.NEXTVAL" />
      <Column name="EVENT_DATE" isEventTimestamp="true" />
      <Column name="LEVEL" pattern="%level" />
      <Column name="LOGGER" pattern="%logger" />
      <Column name="MESSAGE" pattern="%message" />
      <Column name="THROWABLE" pattern="%ex{full}" />
    </JDBC>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
package net.example.db;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;

import javax.sql.DataSource;
import org.apache.commons.dbcp.DriverManagerConnectionFactory;
import org.apache.commons.dbcp.PoolableConnection;
import org.apache.commons.dbcp.PoolableConnectionFactory;
import org.apache.commons.dbcp.PoolingDataSource;
import org.apache.commons.pool.impl.GenericObjectPool;

public class ConnectionFactory {
    private static interface Singleton {
        final ConnectionFactory INSTANCE = new ConnectionFactory();
    }

    private final DataSource dataSource;

    private ConnectionFactory() {
        Properties properties = new Properties();
        properties.setProperty("user", "logging");
        properties.setProperty("password", "abc123"); // or get properties from some configuration file

        GenericObjectPool<PoolableConnection> pool = new GenericObjectPool<PoolableConnection>();
        DriverManagerConnectionFactory connectionFactory = new DriverManagerConnectionFactory("jdbc:mysql://example.org:3306/exampleDb", properties);
        new PoolableConnectionFactory(
            connectionFactory, pool, null, "SELECT 1", 3, false, false, Connection.TRANSACTION_READ_COMMITTED);

        this.dataSource = new PoolingDataSource(pool);
    }

    public static Connection getDatabaseConnection() throws SQLException {
        return Singleton.INSTANCE.dataSource.getConnection();
    }
}

This appender is MapMessage-aware.

The following configuration uses a MessageLayout to indicate that the Appender should match the keys of a MapMessage to the names of ColumnMappings when setting the values of the Appender’s SQL INSERT statement. This let you insert rows for custom values in a database table based on a Log4j MapMessage instead of values from LogEvents.
<Configuration status="debug">

<Appenders>
  <Console name="STDOUT">
    <PatternLayout pattern="%C{1.} %m %level MDC%X%n"/>
  </Console>
  <Jdbc name="databaseAppender" tableName="dsLogEntry" ignoreExceptions="false">
    <DataSource jndiName="java:/comp/env/jdbc/TestDataSourceAppender"/>
    <ColumnMapping name="Id"/>
    <ColumnMapping name="ColumnA"/>
    <ColumnMapping name="ColumnB"/>
    <MessageLayout/>
  </Jdbc>
</Appenders>

<Loggers>
  <Logger name="org.apache.logging.log4j.core.appender.db" level="debug" additivity="false">
    <AppenderRef ref="databaseAppender"/>
  </Logger>
  <Root level="fatal">
    <AppenderRef ref="STDOUT"/>
  </Root>
</Loggers>
</Configuration>

9.1.8 JMS Appender

The JMS Appender sends the formatted log event to a JMS Destination.

Note that in Log4j 2.0, this appender was split into a JMSQueueAppender and a JMSTopicAppender. Starting in Log4j 2.1, these appenders were combined into the JMS Appender which makes no distinction between queues and topics. However, configurations written for 2.0 which use the <JMSQueue/> or <JMSTopic/> elements will continue to work with the new <JMS/> configuration element.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factoryBindingName</td>
<td>String</td>
<td>Required</td>
<td>The name to locate in the Context that provides the ConnectionFactory. This can be any subinterface of ConnectionFactory as well.</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>factoryName</code></td>
<td>String</td>
<td>Required</td>
<td>The fully qualified class name that should be used to define the Initial Context Factory as defined in <code>INITIAL_CONTEXT_FACTORY</code>. If a <code>factoryName</code> is specified without a <code>providerURL</code> a warning message will be logged as this is likely to cause problems.</td>
</tr>
<tr>
<td><code>filter</code></td>
<td>Filter</td>
<td>null</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td><code>layout</code></td>
<td>Layout</td>
<td>Required</td>
<td>The Layout to use to format the LogEvent. New since 2.9, in previous versions <code>SerializedLayout</code> was default.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>String</td>
<td>Required</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td><code>password</code></td>
<td>String</td>
<td>null</td>
<td>The password to use to create the JMS connection.</td>
</tr>
<tr>
<td><code>providerURL</code></td>
<td>String</td>
<td>Required</td>
<td>The URL of the provider to use as defined by <code>PROVIDER_URL</code>.</td>
</tr>
<tr>
<td><code>destinationBindingName</code></td>
<td>String</td>
<td>Required</td>
<td>The name to use to locate the Destination. This can be a Queue or Topic, and as such, the attribute names <code>queueBindingName</code> and <code>topicBindingName</code> are aliases to maintain compatibility with the Log4j 2.0 JMS appenders.</td>
</tr>
<tr>
<td><code>securityPrincipalName</code></td>
<td>String</td>
<td>null</td>
<td>The name of the identity of the Principal as specified by <code>SECURITY_PRINCIPAL</code>. If a <code>securityPrincipalName</code> is specified without <code>securityCredentials</code> a warning message will be logged as this is likely to cause problems.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>securityCredentials</td>
<td>String</td>
<td>null</td>
<td>The security credentials for the principal as specified by SECURITY_CREDENTIALS.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>true</td>
<td>When true, exceptions caught while appending events are internally logged and then ignored. When false exceptions are propagated to the caller. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
<tr>
<td>immediateFail</td>
<td>boolean</td>
<td>false</td>
<td>When set to true, log events will not wait to try to reconnect and will fail immediately if the JMS resources are not available. New in 2.9.</td>
</tr>
<tr>
<td>reconnectIntervalMillis</td>
<td>long</td>
<td>5000</td>
<td>If set to a value greater than 0, after an error, the JMSManager will attempt to reconnect to the broker after waiting the specified number of milliseconds. If the reconnect fails then an exception will be thrown (which can be caught by the application if ignoreExceptions is set to false). New in 2.9.</td>
</tr>
<tr>
<td>urlPkgPrefixes</td>
<td>String</td>
<td>null</td>
<td>A colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory as defined by URL_PKG_PREFIXES.</td>
</tr>
<tr>
<td>userName</td>
<td>String</td>
<td>null</td>
<td>The user id used to create the JMS connection.</td>
</tr>
</tbody>
</table>

**JMS Appender Parameters**

Here is a sample JMS Appender configuration:
To map your Log4j `MapMessage` to JMS `javax.jms.MapMessage`, set the layout of the appender to `MessageLayout` with `<MessageLayout />` (Since 2.9):

```xml
<Configuration status="warn" name="MyApp">
  <Appenders>
    <JMS name="jmsQueue" destinationBindingName="MyQueue"
         factoryBindingName="MyQueueConnectionFactory">
      <MessageLayout />
    </JMS>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="jmsQueue"/>
    </Root>
  </Loggers>
</Configuration>
```

### 9.1.9 JPAAppender

As of Log4j 2.11.0, JPA support has moved from the existing module `logj-core` to the new module `log4j-jpa`.

The JPAAppender writes log events to a relational database table using the Java Persistence API 2.1. It requires the API and a provider implementation be on the classpath. It also requires a decorated entity configured to persist to the table desired. The entity should either extend `org.apache.logging.log4j.core.appender.db.jpa.BasicLogEventEntity` (if you mostly want to use the default mappings) and provide at least an `@Id` property, or `org.apache.logging.log4j.core.appender.db.jpa.AbstractLogEventWrapperEntity` (if you want to significantly customize the mappings). See the Javadoc for these two classes for more information. You can also consult the source code of these two classes as an example of how to implement the entity.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Required. The name of the Appender.</td>
</tr>
</tbody>
</table>
ignoreExceptions  boolean  The default is `true`, causing exceptions encountered while appending events to be internally logged and then ignored. When set to `false` exceptions will be propagated to the caller, instead. You must set this to `false` when wrapping this Appender in a `FailoverAppender`.

filter  Filter  A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a `CompositeFilter`.

bufferSize  int  If an integer greater than 0, this causes the appender to buffer log events and flush whenever the buffer reaches this size.

databaseName  String  Required. The fully qualified name of the concrete `LogEventWrapperEntity` implementation that has JPA annotations mapping it to a database table.

databaseName  String  Required. The name of the JPA persistence unit that should be used for persisting log events.

### JPAAppender Parameters

Here is a sample configuration for the JPAAppender. The first XML sample is the Log4j configuration file, the second is the `persistence.xml` file. EclipseLink is assumed here, but any JPA 2.1 or higher provider will do. You should always create a separate persistence unit for logging, for two reasons. First, `<shared-cache-mode>` must be set to "NONE," which is usually not desired in normal JPA usage. Also, for performance reasons the logging entity should be isolated in its own persistence unit away from all other entities and you should use a non-JTA data source. Note that your persistence unit must also contain `<class>` elements for all of the `org.apache.logging.log4j.core.appender.db.jpa.converter` converter classes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <JPA name="databaseAppender" persistenceUnitName="loggingPersistenceUnit" entityClassName="com.example.logging.JpaLogEntity" />
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
```
<persistence version="2.1">
  <persistence-unit name="loggingPersistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.ContextMapAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.ContextMapJsonAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.ContextStackAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.ContextStackJsonAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.MarkerAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.MessageAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.StackTraceElementAttributeConverter</class>
    <class>org.apache.logging.log4j.core.appender.db.jpa.converter.ThrowableAttributeConverter</class>
    <class>com.example.logging.JpaLogEntity</class>
    <non-jta-data-source>jdbc/LoggingDataSource</non-jta-data-source>
    <shared-cache-mode>NONE</shared-cache-mode>
  </persistence-unit>
</persistence>

package com.example.logging;
...

@Entity
@Table(name="application_log", schema="dbo")
public class JpaLogEntity extends BasicLogEventEntity {
  private static final long serialVersionUID = 1L;
  private long id = 0L;

  public TestEntity() {
    super(null);
  }

  public TestEntity(LogEvent wrappedEvent) {
    super(wrappedEvent);
  }

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "id")
  public long getId() {
    return this.id;
  }

  public void setId(long id) {
    this.id = id;
  }

  // If you want to override the mapping of any properties mapped in BasicLogEventEntity,
  // just override the getters and re-specify the annotations.
}
package com.example.logging;
...

@Entity
@Table(name="application_log", schema="dbo")
public class JpaLogEntity extends AbstractLogEventWrapperEntity {
    private static final long serialVersionUID = 1L;
    private long id = 0L;

    public TestEntity() {
        super(null);
    }

    public TestEntity(LogEvent wrappedEvent) {
        super(wrappedEvent);
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "logEventId")
    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
    }

    @Override
    @Enumerated(EnumType.STRING)
    @Column(name = "level")
    public Level getLevel() {
        return this.getWrappedEvent().getLevel();
    }

    @Override
    @Column(name = "logger")
    public String getLoggerName() {
        return this.getWrappedEvent().getLoggerName();
    }

    @Override
    @Column(name = "message")
    @Convert(converter = MyMessageConverter.class)
    public Message getMessage() {
        return this.getWrappedEvent().getMessage();
    }
    ...
}

9.1.10 HttpAppender

The HttpAppender sends log events over HTTP. A Layout must be provided to format the LogEvent.
Will set the Content-Type header according to the layout. Additional headers can be specified with embedded Property elements.

Will wait for response from server, and throw error if no 2xx response is received.

Implemented with HttpURLConnection.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent.</td>
</tr>
<tr>
<td>Ssl</td>
<td>SslConfiguration</td>
<td>Contains the configuration for the KeyStore and TrustStore for https. Optional, uses Java runtime defaults if not specified. See SSL.</td>
</tr>
<tr>
<td>verifyHostname</td>
<td>boolean</td>
<td>Whether to verify server hostname against certificate. Only valid for https. Optional, defaults to true.</td>
</tr>
<tr>
<td>url</td>
<td>string</td>
<td>The URL to use. The URL scheme must be &quot;http&quot; or &quot;https&quot;.</td>
</tr>
<tr>
<td>method</td>
<td>string</td>
<td>The HTTP method to use. Optional, default is &quot;POST&quot;.</td>
</tr>
<tr>
<td>connectTimeoutMillis</td>
<td>integer</td>
<td>The connect timeout in milliseconds. Optional, default is 0 (infinite timeout).</td>
</tr>
<tr>
<td>readTimeoutMillis</td>
<td>integer</td>
<td>The socket read timeout in milliseconds. Optional, default is 0 (infinite timeout).</td>
</tr>
<tr>
<td>headers</td>
<td>Property[]</td>
<td>Additional HTTP headers to use. The values support lookups.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>

HttpAppender Parameters

Here is a sample HttpAppender configuration snippet:
<xml version="1.0" encoding="UTF-8"?>
  ...
  <Appenders>
    <Http name="Http" url="https://localhost:9200/test/log4j/">
      <Property name="X-Java-Runtime" value="${java:runtime}" />
      <JsonLayout properties="true"/>
      <SSL>
        <KeyStore location="log4j2-keystore.jks" passwordEnvironmentVariable="KEYSTORE_PASSWORD"/>
        <TrustStore location="truststore.jks" passwordFile="${sys:user.home}/truststore.pwd"/>
      </SSL>
    </Http>
  </Appenders>

9.1.11 KafkaAppender

The KafkaAppender logs events to an Apache Kafka topic. Each log event is sent as a Kafka record.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic</td>
<td>String</td>
<td>The Kafka topic to use. Required.</td>
</tr>
<tr>
<td>key</td>
<td>String</td>
<td>The key that will be sent to Kafka with every message. Optional value defaulting to null. Any of the Lookups can be included.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. Required, there is no default. New since 2.9, in previous versions &lt;PatternLayout pattern=&quot;%m&quot;/&gt; was default.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender. Required.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>
### syncSend

**boolean**

The default is `true`, causing sends to block until the record has been acknowledged by the Kafka server. When set to `false` sends return immediately, allowing for lower latency and significantly higher throughput. New since 2.8. Be aware that this is a new addition, and it has not been extensively tested. Any failure sending to Kafka will be reported as error to StatusLogger and the log event will be dropped (the `ignoreExceptions` parameter will not be effective). Log events may arrive out of order to the Kafka server.

### properties

Property[]

You can set properties in Kafka producer properties. You need to set the `bootstrap.servers` property, there are sensible default values for the others. Do not set the `value.serializer` nor `key.serializer` properties.

---

## KafkaAppender Parameters

Here is a sample KafkaAppender configuration snippet:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<Loggers>
  <Root level="DEBUG">
    <AppenderRef ref="Kafka"/>
    <Logger name="org.apache.kafka" level="INFO" /> <!-- avoid recursive logging -->
  </Root>
</Loggers>
```

This appender is synchronous by default and will block until the record has been acknowledged by the Kafka server, timeout for this can be set with the `timeout.ms` property (defaults to 30 seconds). Wrap with Async appender and/or set `syncSend` to `false` to log asynchronously.

This appender requires the Kafka client library. Note that you need to use a version of the Kafka client library matching the Kafka server used.

**Note:** Make sure to not let `org.apache.kafka` log to a Kafka appender on DEBUG level, since that will cause recursive logging:
9.1.12 MemoryMappedFileAppender

New since 2.1. Be aware that this is a new addition, and although it has been tested on several platforms, it does not have as much track record as the other file appenders.

The MemoryMappedFileAppender maps a part of the specified file into memory and writes log events to this memory, relying on the operating system's virtual memory manager to synchronize the changes to the storage device. The main benefit of using memory mapped files is I/O performance. Instead of making system calls to write to disk, this appender can simply change the program's local memory, which is orders of magnitude faster. Also, in most operating systems the memory region mapped actually is the kernel's page cache (file cache), meaning that no copies need to be created in user space. (TODO: performance tests that compare performance of this appender to RandomAccessFileAppender and FileAppender.)

There is some overhead with mapping a file region into memory, especially very large regions (half a gigabyte or more). The default region size is 32 MB, which should strike a reasonable balance between the frequency and the duration of remap operations. (TODO: performance test remapping various sizes.)

Similar to the FileAppender and the RandomAccessFileAppender, MemoryMappedFileAppender uses a MemoryMappedFileManager to actually perform the file I/O. While MemoryMappedFileAppender from different Configurations cannot be shared, the MemoryMappedFileManager can be if the Manager is accessible. For example, two web applications in a servlet container can have their own configuration and safely write to the same file if Log4j is in a ClassLoader that is common to both of them.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>boolean</td>
<td>When true - the default, records will be appended to the end of the file. When set to false, the file will be cleared before new records are written.</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.</td>
</tr>
<tr>
<td>filters</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>immediateFlush</td>
<td>boolean</td>
<td>When set to true, each write will be followed by a call to MappedByteBuffer.force(). This will guarantee the data is written to the storage device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default for this parameter is false. This means that the data is written to the storage device even if the Java process crashes, but there may be data loss if the operating system crashes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that manually forcing a sync on every log event loses most of the performance benefits of using a memory mapped file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flushing after every write is only useful when using this appender with synchronous loggers. Asynchronous loggers and appenders will automatically flush at the end of a batch of events, even if immediateFlush is set to false. This also guarantees the data is written to disk but is more efficient.</td>
</tr>
<tr>
<td>regionLength</td>
<td>int</td>
<td>The length of the mapped region, defaults to 32 MB (32 * 1024 * 1024 bytes). This parameter must be a value between 256 and 1,073,741,824 (1 GB or 2^30); values outside this range will be adjusted to the closest valid value. Log4j will round the specified value up to the nearest power of two.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of &quot;%m%n&quot; will be used.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>
Here is a sample MemoryMappedFile configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <MemoryMappedFile name="MyFile" fileName="logs/app.log">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
    </MemoryMappedFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="MyFile"/>
    </Root>
  </Loggers>
</Configuration>
```

### 9.1.13 NoSQLAppender

The NoSQLAppender writes log events to a NoSQL database using an internal lightweight provider interface. Provider implementations currently exist for MongoDB and Apache CouchDB, and writing a custom provider is quite simple.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Required. The name of the Appender.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>bufferSize</td>
<td>int</td>
<td>If an integer greater than 0, this causes the appender to buffer log events and flush whenever the buffer reaches this size.</td>
</tr>
<tr>
<td>NoSqlProvider</td>
<td>NoSQLProvider&lt;C extends NoSQLConnection&lt;W, T extends NoSQLObject&lt;W&gt;&gt;&gt;</td>
<td>Required. The NoSQL provider that provides connections to the chosen NoSQL database.</td>
</tr>
</tbody>
</table>

NoSQLAppender Parameters
You specify which NoSQL provider to use by specifying the appropriate configuration element within the `<NoSql>` element. The types currently supported are `<MongoDb>` and `<CouchDb>`. To create your own custom provider, read the JavaDoc for the `NoSQLProvider`, `NoSQLConnection`, and `NoSQLObject` classes and the documentation about creating Log4j plugins. We recommend you review the source code for the MongoDB and CouchDB providers as a guide for creating your own provider.

The following example demonstrates how log events are persisted in NoSQL databases if represented in a JSON format:
{
  "level": "WARN",
  "loggerName": "com.example.application.MyClass",
  "message": "Something happened that you might want to know about.",
  "source": {
    "className": "com.example.application.MyClass",
    "methodName": "exampleMethod",
    "fileName": "MyClass.java",
    "lineNumber": 81
  },
  "marker": {
    "name": "SomeMarker",
    "parent": {
      "name": "SomeParentMarker"
    }
  },
  "threadName": "Thread-1",
  "millis": 136884166761,
  "date": "2013-05-18T02:29:26.761Z",
  "thrown": {
    "type": "java.sql.SQLException",
    "message": "Could not insert record. Connection lost.",
    "stackTrace": [
      { "className": "org.example.sql.driver.PreparedStatement$1", "methodName": "responder", "fileName": "PreparedStatement.java", "lineNumber": 1049 },
      { "className": "org.example.sql.driver.PreparedStatement", "methodName": "executeUpdate", "fileName": "PreparedStatement.java", "lineNumber": 738 },
      { "className": "com.example.application.MyClass", "methodName": "exampleMethod", "fileName": "MyClass.java", "lineNumber": 81 },
      { "className": "com.example.application.MainClass", "methodName": "main", "fileName": "MainClass.java", "lineNumber": 52 }
    ],
    "cause": {
      "type": "java.io.IOException",
      "message": "Connection lost.",
      "stackTrace": [
        { "className": "java.nio.channels.SocketChannel", "methodName": "write", "fileName": null, "lineNumber": -1 },
        { "className": "org.example.sql.driver.PreparedStatement$1", "methodName": "responder", "fileName": "PreparedStatement.java", "lineNumber": 1032 },
        { "className": "org.example.sql.driver.PreparedStatement", "methodName": "executeUpdate", "fileName": "PreparedStatement.java", "lineNumber": 738 },
        { "className": "com.example.application.MyClass", "methodName": "exampleMethod", "fileName": "MyClass.java", "lineNumber": 81 },
        { "className": "com.example.application.MainClass", "methodName": "main", "fileName": "MainClass.java", "lineNumber": 52 }
      ]
    }
  },
  "contextMap": {
    "ID": "86c3a497-4e67-4eed-9d6a-2e5797324d7b",
    "username": "JohnDoe"
  },
  "contextStack": [
    "topItem",
    "anotherItem",
    "bottomItem"
  ]
}
### 9.1.14 NoSQLAppender for MongoDB

Starting with Log4J 2.11.0, we provide two MongoDB modules:

- `log4j-mongodb2` defines the configuration element `MongoDb2` matching the MongoDB Driver version 2.
- `log4j-mongodb3` defines the configuration element `MongoDb3` matching the MongoDB Driver version 3.

We no longer provide the module `log4j-mongodb`.

The module `log4j-mongodb2` aliases the old configuration element `MongoDb` to `MongoDb2`.

### 9.1.15 NoSQLAppender for MongoDB 2

This section details specializations of the NoSQLAppender provider for MongoDB using the MongoDB driver version 2. The NoSQLAppender Appender writes log events to a NoSQL database using an internal lightweight provider interface.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collectionName</td>
<td>String</td>
<td>Required. The name of the MongoDB collection to insert the events into.</td>
</tr>
<tr>
<td>writeConcernConstant</td>
<td>Field</td>
<td>By default, the MongoDB provider inserts records with the instructions <code>com.mongodb.WriteConcern.ACKNOWLEDGED</code>. Use this optional attribute to specify the name of a constant other than <code>ACKNOWLEDGED</code>.</td>
</tr>
<tr>
<td>writeConcernConstantClass</td>
<td>Class</td>
<td>If you specify <code>writeConcernConstant</code>, you can use this attribute to specify a class other than <code>com.mongodb.WriteConcern</code> to find the constant on (to create your own custom instructions).</td>
</tr>
<tr>
<td>factoryClassName</td>
<td>Class</td>
<td>To provide a connection to the MongoDB database, you can use this attribute and <code>factoryMethodName</code> to specify a class and static method to get the connection from. The method must return a <code>com.mongodb.DB</code> or a <code>com.mongodb.MongoClient</code>. If the DB is not authenticated, you must also specify a username and password. If you use the factory method for providing a connection, you must not specify the <code>databaseName</code>, <code>server</code>, or <code>port</code> attributes.</td>
</tr>
<tr>
<td>factoryMethodName</td>
<td>Method</td>
<td>See the documentation for attribute <code>factoryClassName</code>.</td>
</tr>
</tbody>
</table>
MongoDB2 Provider Parameters

This appender is MapMessage-aware.

Here are a few sample configurations for the NoSQLAppender and MongoDB2 provider:

```xml
<Configuration status="error">
  <Appenders>
    <NoSql name="databaseAppender">
      <MongoDb2 databaseName="applicationDb" collectionName="applicationLog" server="mongo.example.org" username="loggingUser" password="abc123" />
    </NoSql>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
```
Starting in Log4j version 2.11.0, the provider element name is `MongoDb2`. The name `MongoDb` is now a deprecated alias for `MongoDb2`.

### 9.1.16 NoSQLAppender for MongoDB 3

This section details specializations of the `NoSQLAppender` provider for MongoDB using the MongoDB driver version 3. The NoSQLAppender Appender writes log events to a NoSQL database using an internal lightweight provider interface.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collectionName</code></td>
<td>String</td>
<td>Required. The name of the MongoDB collection to insert the events into.</td>
</tr>
<tr>
<td><code>writeConcernConstant</code></td>
<td>Field</td>
<td>By default, the MongoDB provider inserts records with the instructions <code>com.mongodb.WriteConcern.ACKNOWLEDGED</code>. Use this optional attribute to specify the name of a constant other than <code>ACKNOWLEDGED</code>.</td>
</tr>
<tr>
<td><code>writeConcernConstantClass</code></td>
<td>Class</td>
<td>If you specify <code>writeConcernConstant</code>, you can use this attribute to specify a class other than <code>com.mongodb.WriteConcern</code> to find the constant on (to create your own custom instructions).</td>
</tr>
<tr>
<td><strong>factoryClassName</strong></td>
<td><strong>Class</strong></td>
<td>To provide a connection to the MongoDB database, you can use this attribute and factoryMethodName to specify a class and static method to get the connection from. The method must return a com.mongodb.client.MongoDatabase or a com.mongodb.MongoClient. If the com.mongodb.client.MongoDatabase is not authenticated, you must also specify a username and password. If you use the factory method for providing a connection, you must not specify the databaseName, server, or port attributes.</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td><strong>factoryMethodName</strong></td>
<td><strong>Method</strong></td>
<td>See the documentation for attribute factoryClassName.</td>
</tr>
<tr>
<td><strong>databaseName</strong></td>
<td><strong>String</strong></td>
<td>If you do not specify a factoryClassName and factoryMethodName for providing a MongoDB connection, you must specify a MongoDB database name using this attribute. You must also specify a username and password. You can optionally also specify a server (defaults to localhost), and a port (defaults to the default MongoDB port).</td>
</tr>
<tr>
<td><strong>server</strong></td>
<td><strong>String</strong></td>
<td>See the documentation for attribute databaseName.</td>
</tr>
<tr>
<td><strong>port</strong></td>
<td><strong>int</strong></td>
<td>See the documentation for attribute databaseName.</td>
</tr>
<tr>
<td><strong>username</strong></td>
<td><strong>String</strong></td>
<td>See the documentation for attributes databaseName and factoryClassName.</td>
</tr>
<tr>
<td><strong>password</strong></td>
<td><strong>String</strong></td>
<td>See the documentation for attributes databaseName and factoryClassName.</td>
</tr>
<tr>
<td><strong>capped</strong></td>
<td><strong>boolean</strong></td>
<td>Enable support for capped collections</td>
</tr>
<tr>
<td><strong>collectionSize</strong></td>
<td><strong>int</strong></td>
<td>Specify the size in bytes of the capped collection to use if enabled. The minimum size is 4096 bytes, and larger sizes will be increased to the nearest integer multiple of 256. See the capped collection documentation linked above for more information.</td>
</tr>
</tbody>
</table>
MongoDB3 Provider Parameters

This appender is MapMessage-aware.

Here are a few sample configurations for the NoSQLAppender and MongoDB3 provider:

```xml
<Configuration status="error">
  <Appenders>
    <NoSql name="databaseAppender">
      <MongoDb3 databaseName="applicationDb" collectionName="applicationLog" server="mongo.example.org"
        username="loggingUser" password="abc123" />
    </NoSql>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

```xml
<Configuration status="error">
  <Appenders>
    <NoSql name="databaseAppender">
      <MongoDb3 collectionName="applicationLog" factoryClassName="org.example.db.ConnectionFactory"
        factoryMethodName="getNewMongoClient" />
    </NoSql>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

### 9.1.17 NoSQLAppender for Apache CouchDB

This section details specializations of the NoSQLAppender provider for CouchDB. The NoSQLAppender writes log events to a NoSQL database using an internal lightweight provider interface.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>

© 2019, The Apache Software Foundation • ALL RIGHTS RESERVED.
CouchDB Provider Parameters

Here are a few sample configurations for the NoSQLAppender and CouchDB provider:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factoryClassName</td>
<td>Class</td>
<td>To provide a connection to the CouchDB database, you can use this attribute and factoryMethodName to specify a class and static method to get the connection from. The method must return a org.lightcouch.CouchDbClient or a org.lightcouch.CouchDbProperties. If you use the factory method for providing a connection, you must not specify the databaseName, protocol, server, port, username, or password attributes.</td>
</tr>
<tr>
<td>factoryMethodName</td>
<td>Method</td>
<td>See the documentation for attribute factoryClassName.</td>
</tr>
<tr>
<td>databaseName</td>
<td>String</td>
<td>If you do not specify a factoryClassName and factoryMethodName for providing a CouchDB connection, you must specify a CouchDB database name using this attribute. You must also specify a username and password. You can optionally also specify a protocol (defaults to http), server (defaults to localhost), and a port (defaults to 80 for http and 443 for https).</td>
</tr>
<tr>
<td>protocol</td>
<td>String</td>
<td>Must either be &quot;http&quot; or &quot;https.&quot; See the documentation for attribute databaseName.</td>
</tr>
<tr>
<td>server</td>
<td>String</td>
<td>See the documentation for attribute databaseName.</td>
</tr>
<tr>
<td>port</td>
<td>int</td>
<td>See the documentation for attribute databaseName.</td>
</tr>
<tr>
<td>username</td>
<td>String</td>
<td>See the documentation for attributes databaseName.</td>
</tr>
<tr>
<td>password</td>
<td>String</td>
<td>See the documentation for attributes databaseName.</td>
</tr>
</tbody>
</table>
9.1.18 OutputStreamAppender
The OutputStreamAppender provides the base for many of the other Appenders such as the File and Socket appenders that write the event to an Output Stream. It cannot be directly configured. Support for immediateFlush and buffering is provided by the OutputStreamAppender. The OutputStreamAppender uses an OutputStreamManager to handle the actual I/O, allowing the stream to be shared by Appenders in multiple configurations.

9.1.19 RandomAccessFileAppender
The RandomAccessFileAppender is similar to the standard FileAppender except it is always buffered (this cannot be switched off) and internally it uses a ByteBuffer + RandomAccessFile instead of a BufferedOutputStream. We saw a 20-200% performance improvement compared to FileAppender with "bufferedIO=true" in our measurements. Similar to the FileAppender, RandomAccessFileAppender uses a RandomAccessFileManager to actually perform the file I/O. While RandomAccessFileAppender from different Configurations cannot be shared, the RandomAccessFileManager can be if the Manager is accessible. For example, two web applications in a servlet container can have their own configuration and safely write to the same file if Log4j is in a ClassLoader that is common to both of them.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>boolean</td>
<td>When true - the default, records will be appended to the end of the file. When set to false, the file will be cleared before new records are written.</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.</td>
</tr>
<tr>
<td>filters</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
</tbody>
</table>
**immediateFlush**  
boolean  
When set to true - the default, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance.

Flushing after every write is only useful when using this appender with synchronous loggers. Asynchronous loggers and appenders will automatically flush at the end of a batch of events, even if immediateFlush is set to false. This also guarantees the data is written to disk but is more efficient.

**bufferSize**  
int  
The buffer size, defaults to 262,144 bytes (256 * 1024).

**layout**  
Layout  
The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of "%m%n" will be used.

**name**  
String  
The name of the Appender.

**ignoreExceptions**  
boolean  
The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.

---

**RandomAccessFileAppender Parameters**

Here is a sample RandomAccessFile configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RandomAccessFile name="MyFile" fileName="logs/app.log">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
    </RandomAccessFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="MyFile"/>
    </Root>
  </Loggers>
</Configuration>
```
9.1.20 RewriteAppender

The RewriteAppender allows the LogEvent to manipulated before it is processed by another Appender. This can be used to mask sensitive information such as passwords or to inject information into each event. The RewriteAppender must be configured with a RewritePolicy. The RewriteAppender should be configured after any Appenders it references to allow it to shut down properly.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppenderRef</td>
<td>String</td>
<td>The name of the Appenders to call after the LogEvent has been manipulated. Multiple AppenderRef elements can be configured.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>rewritePolicy</td>
<td>RewritePolicy</td>
<td>The RewritePolicy that will manipulate the LogEvent.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>

RewriteAppender Parameters

9.1.20.1 RewritePolicy

RewritePolicy is an interface that allows implementations to inspect and possibly modify LogEvents before they are passed to Appender. RewritePolicy declares a single method named rewrite that must be implemented. The method is passed the LogEvent and can return the same event or create a new one.

9.MapRewritePolicy

MapRewritePolicy will evaluate LogEvents that contain a MapMessage and will add or update elements of the Map.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>String</td>
<td>&quot;Add&quot; or &quot;Update&quot;</td>
</tr>
<tr>
<td>keyValuePair</td>
<td>KeyValuePair[]</td>
<td>An array of keys and their values.</td>
</tr>
</tbody>
</table>

The following configuration shows a RewriteAppender configured to add a product key and its value to the MapMessage.:
9. Properties RewritePolicy

Properties RewritePolicy will add properties configured on the policy to the ThreadContext Map being logged. The properties will not be added to the actual ThreadContext Map. The property values may contain variables that will be evaluated when the configuration is processed as well as when the event is logged.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties</td>
<td>Property[]</td>
<td>One of more Property elements to define the keys and values to be added to the ThreadContext Map.</td>
</tr>
</tbody>
</table>

The following configuration shows a RewriteAppender configured to add a product key and its value to the MapMessage:
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%m%n"/>
    </Console>
    <Rewrite name="rewrite">
      <AppenderRef ref="STDOUT"/>
      <PropertiesRewritePolicy>
        <Property name="user">${sys:user.name}</Property>
        <Property name="env">${sys:environment}</Property>
      </PropertiesRewritePolicy>
    </Rewrite>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Rewrite"/>
    </Root>
  </Loggers>
</Configuration>

9. LoggerNameLevelRewritePolicy

You can use this policy to make loggers in third party code less chatty by changing event levels. The LoggerNameLevelRewritePolicy will rewrite log event levels for a given logger name prefix. You configure a LoggerNameLevelRewritePolicy with a logger name prefix and a pairs of levels, where a pair defines a source level and a target level.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logger</td>
<td>String</td>
<td>A logger name used as a prefix to test each event's logger name.</td>
</tr>
<tr>
<td>LevelPair</td>
<td>KeyValuePair[]</td>
<td>An array of keys and their values, each key is a source level, each value a target level.</td>
</tr>
</tbody>
</table>

The following configuration shows a RewriteAppender configured to map level INFO to DEBUG and level WARN to INFO for all loggers that start with `com.foo.bar`.
9.1.21 RollingFileAppender

The RollingFileAppender is an OutputStreamAppender that writes to the File named in the fileName parameter and rolls the file over according the TriggeringPolicy and the RolloverPolicy. The RollingFileAppender uses a RollingFileManager (which extends OutputStreamManager) to actually perform the file I/O and perform the rollover. While RolloverFileAppenders from different Configurations cannot be shared, the RollingFileManager can be if the Manager is accessible. For example, two web applications in a servlet container can have their own configuration and safely write to the same file if Log4j is in a ClassLoader that is common to both of them.

A RollingFileAppender requires a TriggeringPolicy and a RolloverStrategy. The triggering policy determines if a rollover should be performed while the RolloverStrategy defines how the rollover should be done. If no RolloverStrategy is configured, RollingFileAppender will use the DefaultRolloverStrategy. Since log4j-2.5, a custom delete action can be configured in the DefaultRolloverStrategy to run at rollover. Since 2.8 if no file name is configured then DirectWriteRolloverStrategy will be used instead of DefaultRolloverStrategy. Since log4j-2.9, a custom POSIX file attribute view action can be configured in the DefaultRolloverStrategy to run at rollover, if not defined, inherited POSIX file attribute view from the RollingFileAppender will be applied.

File locking is not supported by the RollingFileAppender.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>boolean</td>
<td>When true - the default, records will be appended to the end of the file. When set to false, the file will be cleared before new records are written.</td>
</tr>
<tr>
<td><strong>bufferedIO</strong></td>
<td>boolean</td>
<td>When true - the default, records will be written to a buffer and the data will be written to disk when the buffer is full or, if immediateFlush is set, when the record is written. File locking cannot be used with bufferedIO. Performance tests have shown that using buffered I/O significantly improves performance, even if immediateFlush is enabled.</td>
</tr>
<tr>
<td><strong>bufferSize</strong></td>
<td>int</td>
<td>When bufferedIO is true, this is the buffer size, the default is 8192 bytes.</td>
</tr>
<tr>
<td><strong>createOnDemand</strong></td>
<td>boolean</td>
<td>The appender creates the file on-demand. The appender only creates the file when a log event passes all filters and is routed to this appender. Defaults to false.</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td><strong>fileName</strong></td>
<td>String</td>
<td>The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.</td>
</tr>
<tr>
<td><strong>filePattern</strong></td>
<td>String</td>
<td>The pattern of the file name of the archived log file. The format of the pattern is dependent on the RolloverPolicy that is used. The DefaultRolloverPolicy will accept both a date/time pattern compatible with SimpleDateFormat and/or a %i which represents an integer counter. The pattern also supports interpolation at runtime so any of the Lookups (such as the DateLookup) can be included in the pattern.</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>immediateFlush</td>
<td>boolean</td>
<td>When set to true - the default, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance. Flushing after every write is only useful when using this appender with synchronous loggers. Asynchronous loggers and appenders will automatically flush at the end of a batch of events, even if immediateFlush is set to false. This also guarantees the data is written to disk but is more efficient.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of &quot;%m%n&quot; will be used.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>policy</td>
<td>TriggeringPolicy</td>
<td>The policy to use to determine if a rollover should occur.</td>
</tr>
<tr>
<td>strategy</td>
<td>RolloverStrategy</td>
<td>The strategy to use to determine the name and location of the archive file.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
<tr>
<td>filePermissions</td>
<td>String</td>
<td>File attribute permissions in POSIX format to apply whenever the file is created. Underlying files system shall support POSIX file attribute view. Examples: rw------- or rw-rw- rw- etc...</td>
</tr>
</tbody>
</table>
### RollingFileAppender Parameters

#### 9.1.21.1 Triggering Policies

**9. Composite Triggering Policy**

The `CompositeTriggeringPolicy` combines multiple triggering policies and returns true if any of the configured policies return true. The `CompositeTriggeringPolicy` is configured simply by wrapping other policies in a `Policies` element.

For example, the following XML fragment defines policies that rollover the log when the JVM starts, when the log size reaches twenty megabytes, and when the current date no longer matches the log’s start date.

```xml
<Policies>
	<OnStartupTriggeringPolicy />
	<SizeBasedTriggeringPolicy size="20 MB" />
	<TimeBasedTriggeringPolicy />
</Policies>
```

**9. Cron Triggering Policy**

The `CronTriggeringPolicy` triggers rollover based on a cron expression. This policy is controlled by a timer and is asynchronous to processing log events, so it is possible that log events from the previous or next time period may appear at the beginning or end of the log file. The `filePattern` attribute of the Appender should contain a timestamp otherwise the target file will be overwritten on each rollover.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileOwner</td>
<td>String</td>
<td>File owner to define whenever the file is created. Changing file’s owner may be restricted for security reason and Operation not permitted IOException thrown. Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file if <code>_POSIX_CHOWN_RESTRICTED</code> is in effect for path. Underlying files system shall support file owner attribute view.</td>
</tr>
<tr>
<td>fileGroup</td>
<td>String</td>
<td>File group to define whenever the file is created. Underlying files system shall support POSIX file attribute view.</td>
</tr>
</tbody>
</table>
schedule String
The cron expression. The expression is the same as what is allowed in the Quartz scheduler. See CronExpression for a full description of the expression.
evaluateOnStartup boolean
On startup the cron expression will be evaluated against the file's last modification timestamp. If the cron expression indicates a rollover should have occurred between that time and the current time the file will be immediately rolled over.

CronTriggeringPolicy Parameters

9. OnStartup Triggering Policy
The OnStartupTriggeringPolicy policy causes a rollover if the log file is older than the current JVM's start time and the minimum file size is met or exceeded.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minSize</td>
<td>long</td>
<td>The minimum size the file must have to roll over. A size of zero will cause a roll over no matter what the file size is. The default value is 1, which will prevent rolling over an empty file.</td>
</tr>
</tbody>
</table>

OnStartupTriggeringPolicy Parameters

Google App Engine note:
When running in Google App Engine, the OnStartup policy causes a rollover if the log file is older than the time when Log4J initialized. (Google App Engine restricts access to certain classes so Log4J cannot determine JVM start time with java.lang.management.ManagementFactory.getRuntimeMXBean().getStartTime() and falls back to Log4J initialization time instead.)

9. SizeBased Triggering Policy
The SizeBasedTriggeringPolicy causes a rollover once the file has reached the specified size. The size can be specified in bytes, with the suffix KB, MB or GB, for example 20MB. The file pattern must contain a %i otherwise the target file will be overwritten on every rollover as the SizeBased Triggering Policy does not cause the timestamp value in the file name to change.

9. TimeBased Triggering Policy
The TimeBasedTriggeringPolicy causes a rollover once the date/time pattern no longer applies to the active file. This policy accepts an interval attribute which indicates how frequently the rollover should occur based on the time pattern and a modulate boolean attribute.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
### TimeBasedTriggeringPolicy Parameters

#### 9.1.21.2 Rollover Strategies

#### 9. Default Rollover Strategy

The default rollover strategy accepts both a date/time pattern and an integer from the filePattern attribute specified on the RollingFileAppender itself. If the date/time pattern is present it will be replaced with the current date and time values. If the pattern contains an integer it will be incremented on each rollover. If the pattern contains both a date/time and integer in the pattern the integer will be incremented until the result of the date/time pattern changes. 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 formats bzip2, Deflate, Pack200 and XZ require Apache Commons Compress. In addition, XZ requires XZ for Java. The pattern may also contain lookup references that can be resolved at runtime such as is shown in the example below.

The default rollover strategy supports three variations for incrementing the counter. The first is the "fixed window" strategy. To illustrate how it works, suppose that the min attribute is set to 1, the max attribute is set to 3, the file name is "foo.log", and the file name pattern is "foo-%i.log".

<table>
<thead>
<tr>
<th>Number of rollovers</th>
<th>Active output target</th>
<th>Archived log files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>foo.log</td>
<td>-</td>
<td>All logging is going to the initial file.</td>
</tr>
</tbody>
</table>

- interval
  - **integer**
  - How often a rollover should occur based on the most specific time unit in the date pattern. For example, with a date pattern with hours as the most specific item and increment of 4 rollovers would occur every 4 hours. The default value is 1.
- modulate
  - **boolean**
  - Indicates whether the interval should be adjusted to cause the next rollover to occur on the interval boundary. For example, if the item is hours, the current hour is 3 am and the interval is 4 then the first rollover will occur at 4 am and then next ones will occur at 8 am, noon, 4pm, etc.
- maxRandomDelay
  - **integer**
  - Indicates the maximum number of seconds to randomly delay a rollover. By default, this is 0 which indicates no delay. This setting is useful on servers where multiple applications are configured to rollover log files at the same time and can spread the load of doing so across time.
<table>
<thead>
<tr>
<th>Number of rollovers</th>
<th>Active output target</th>
<th>Archived log files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>foo.log</td>
<td>-</td>
<td>All logging is going to the initial file.</td>
</tr>
<tr>
<td>1</td>
<td>foo.log</td>
<td>foo-1.log</td>
<td>During the first rollover foo.log is renamed to foo-1.log. A new foo.log file is created and starts being written to.</td>
</tr>
<tr>
<td>2</td>
<td>foo.log</td>
<td>foo-1.log, foo-2.log</td>
<td>During the second rollover foo-1.log is renamed to foo-2.log and foo.log is renamed to foo-1.log. A new foo.log file is created and starts being written to.</td>
</tr>
<tr>
<td>3</td>
<td>foo.log</td>
<td>foo-1.log, foo-2.log, foo-3.log</td>
<td>During the third rollover foo-2.log is renamed to foo-3.log, foo-1.log is renamed to foo-2.log and foo.log is renamed to foo-1.log. A new foo.log file is created and starts being written to.</td>
</tr>
<tr>
<td>4</td>
<td>foo.log</td>
<td>foo-1.log, foo-2.log, foo-3.log</td>
<td>In the fourth and subsequent rollovers, foo-3.log is deleted, foo-2.log is renamed to foo-3.log, foo-1.log is renamed to foo-2.log and foo.log is renamed to foo-1.log. A new foo.log file is created and starts being written to.</td>
</tr>
</tbody>
</table>

By way of contrast, when the fileIndex attribute is set to "max" but all the other settings are the same the following actions will be performed.
Finally, as of release 2.8, if the fileIndex attribute is set to “nomax” then the min and max values will be ignored and file numbering will increment by 1 and each rollover will have an incrementally higher value with no maximum number of files.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileIndex</td>
<td>String</td>
<td>If set to “max” (the default), files with a higher index will be newer than files with a smaller index. If set to “min”, file renaming and the counter will follow the Fixed Window strategy described above.</td>
</tr>
<tr>
<td>min</td>
<td>integer</td>
<td>The minimum value of the counter. The default value is 1.</td>
</tr>
<tr>
<td>max</td>
<td>integer</td>
<td>The maximum value of the counter. Once this value is reached older archives will be deleted on subsequent rollovers. The default value is 7.</td>
</tr>
<tr>
<td>compressionLevel</td>
<td>integer</td>
<td>Sets the compression level, 0-9, where 0 = none, 1 = best speed, through 9 = best compression. Only implemented for ZIP files.</td>
</tr>
<tr>
<td>tempCompressedFilePattern</td>
<td>String</td>
<td>The pattern of the file name of the archived log file during compression.</td>
</tr>
</tbody>
</table>

DefaultRolloverStrategy Parameters

9. DirectWrite Rollover Strategy
The DirectWriteRolloverStrategy causes log events to be written directly to files represented by the file pattern. With this strategy file renames are not performed. If the size-based triggering policy causes multiple files to be written during the specified time period they will be numbered starting at one and continually incremented until a time-based rollover occurs.

Warning: If the file pattern has a suffix indicating compression should take place the current file will not be compressed when the application is shut down. Furthermore, if the time changes such that the file pattern no longer matches the current file it will not be compressed at startup either.
### DirectWriteRolloverStrategy Parameters

Below is a sample configuration that uses a RollingFileAppender with both the time and size based triggering policies, will create up to 7 archives on the same day (1-7) that are stored in a directory based on the current year and month, and will compress each archive using gzip:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <TimeBasedTriggeringPolicy />
        <SizeBasedTriggeringPolicy size="250 MB" />
      </Policies>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

This second example shows a rollover strategy that will keep up to 20 files before removing them.
<?xml version="1.0" encoding="UTF-8"?><Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <TimeBasedTriggeringPolicy />
        <SizeBasedTriggeringPolicy size="250 MB"/>
      </Policies>
      <DefaultRolloverStrategy max="20"/>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>

Below is a sample configuration that uses a RollingFileAppender with both the time and size based triggering policies, will create up to 7 archives on the same day (1-7) that are stored in a directory based on the current year and month, and will compress each archive using gzip and will roll every 6 hours when the hour is divisible by 6:

<?xml version="1.0" encoding="UTF-8"?><Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/$${date:yyyy-MM}/app-%d{yyyy-MM-dd-HH}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <TimeBasedTriggeringPolicy interval="6" modulate="true"/>
        <SizeBasedTriggeringPolicy size="250 MB"/>
      </Policies>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>

This sample configuration uses a RollingFileAppender with both the cron and size based triggering policies, and writes directly to an unlimited number of archive files. The cron trigger causes a rollover every hour while the file size is limited to 250MB:
This sample configuration is the same as the previous but limits the number of files saved each hour to 10:

```xml
<Configuration status="warn" name="MyApp" packages="">
    <Appenders>
        <RollingFile name="RollingFile" filePattern="logs/app-%d{yyyy-MM-dd-HH}-%i.log.gz">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
            </PatternLayout>
            <Policies>
                <CronTriggeringPolicy schedule="0 0 * * * ?"/>
                <SizeBasedTriggeringPolicy size="250 MB"/>
                <DirectWriteRolloverStrategy maxFiles="10"/>
            </Policies>
        </RollingFile>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="RollingFile"/>
        </Root>
    </Loggers>
</Configuration>
```

9. Log Archive Retention Policy: Delete on Rollover

Log4j-2.5 introduces a Delete action that gives users more control over what files are deleted at rollover time than what was possible with the DefaultRolloverStrategy max attribute. The Delete action lets users configure one or more conditions that select the files to delete relative to a base directory.

Note that it is possible to delete any file, not just rolled over log files, so use this action with care! With the testMode parameter you can test your configuration without accidentally deleting the wrong files.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basePath</td>
<td>String</td>
<td>Required. Base path from where to start scanning for files to delete.</td>
</tr>
<tr>
<td>maxDepth</td>
<td>int</td>
<td>The maximum number of levels of directories to visit. A value of 0 means</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that only the starting file (the base path itself) is visited, unless denied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by the security manager. A value of Integer.MAX_VALUE indicates that all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>levels should be visited. The default is 1, meaning only the files in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specified base directory.</td>
</tr>
<tr>
<td>followLinks</td>
<td>boolean</td>
<td>Whether to follow symbolic links. Default is false.</td>
</tr>
<tr>
<td>testMode</td>
<td>boolean</td>
<td>If true, files are not deleted but instead a message is printed to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>status logger at INFO level. Use this to do a dry run to test if the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>configuration works as expected. Default is false.</td>
</tr>
<tr>
<td>pathSorter</td>
<td>PathSorter</td>
<td>A plugin implementing the PathSorter interface to sort the files before</td>
</tr>
<tr>
<td></td>
<td></td>
<td>selecting the files to delete. The default is to sort most recently modified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>files first.</td>
</tr>
<tr>
<td>pathConditions</td>
<td>PathCondition[]</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td></td>
</tr>
</tbody>
</table>

Required if no `ScriptCondition` is specified. One or more `PathCondition` elements.

If more than one condition is specified, they all need to accept a path before it is deleted. Conditions can be nested, in which case the inner condition(s) are evaluated only if the outer condition accepts the path. If conditions are not nested they may be evaluated in any order.

Conditions can also be combined with the logical operators AND, OR and NOT by using the `IfAll`, `IfAny` and `IfNot` composite conditions.

Users can create custom conditions or use the built-in conditions:

- **IfFileName** - accepts files whose path (relative to the base path) matches a regular expression or a glob.
- **IfLastModified** - accepts files that are as old as or older than the specified duration.
- **IfAccumulatedFileCount** - accepts paths after some count threshold is exceeded during the file tree walk.
- **IfAccumulatedFileSize** - accepts paths after the accumulated file size threshold is exceeded during the file tree walk.
- **IfAll** - accepts a path if all nested conditions accept it (logical AND). Nested conditions may be evaluated in any order.
- **IfAny** - accepts a path if one of the nested conditions accept it (logical OR). Nested conditions may be evaluated in any order.
- **IfNot** - accepts a path if the nested condition does not accept it (logical NOT).
scriptCondition | ScriptCondition | Required if no PathConditions are specified. A ScriptCondition element specifying a script.

The ScriptCondition should contain a Script, ScriptRef or ScriptFile element that specifies the logic to be executed. (See also the ScriptFilter documentation for more examples of configuring ScriptFiles and ScriptRefs.)

The script is passed a number of parameters, including a list of paths found under the base path (up to maxDepth) and must return a list with the paths to delete.

Delete Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>glob</td>
<td>String</td>
<td>Required if regex not specified. Matches the relative path (relative to the base path) using a limited pattern language that resembles regular expressions but with a simpler syntax.</td>
</tr>
<tr>
<td>regex</td>
<td>String</td>
<td>Required if glob not specified. Matches the relative path (relative to the base path) using a regular expression as defined by the Pattern class.</td>
</tr>
<tr>
<td>nestedConditions</td>
<td>PathCondition[]</td>
<td>An optional set of nested PathConditions. If any nested conditions exist they all need to accept the file before it is deleted. Nested conditions are only evaluated if the outer condition accepts a file (if the path name matches).</td>
</tr>
</tbody>
</table>

IfFileName Condition Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
appenders

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>String</td>
<td>Required. Specifies a duration. The condition accepts files that are as old or older than the specified duration.</td>
</tr>
<tr>
<td>nestedConditions</td>
<td>PathCondition[]</td>
<td>An optional set of nested PathConditions. If any nested conditions exist they all need to accept the file before it is deleted. Nested conditions are only evaluated if the outer condition accepts a file (if the file is old enough).</td>
</tr>
</tbody>
</table>

IfLastModified Condition Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exceeds</td>
<td>int</td>
<td>Required. The threshold count from which files will be deleted.</td>
</tr>
<tr>
<td>nestedConditions</td>
<td>PathCondition[]</td>
<td>An optional set of nested PathConditions. If any nested conditions exist they all need to accept the file before it is deleted. Nested conditions are only evaluated if the outer condition accepts a file (if the threshold count has been exceeded).</td>
</tr>
</tbody>
</table>

IfAccumulatedFileCount Condition Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exceeds</td>
<td>String</td>
<td>Required. The threshold accumulated file size from which files will be deleted. The size can be specified in bytes, with the suffix KB, MB or GB, for example 20MB.</td>
</tr>
<tr>
<td>nestedConditions</td>
<td>PathCondition[]</td>
<td>An optional set of nested PathConditions. If any nested conditions exist they all need to accept the file before it is deleted. Nested conditions are only evaluated if the outer condition accepts a file (if the threshold accumulated file size has been exceeded).</td>
</tr>
</tbody>
</table>

IfAccumulatedFileSize Condition Parameters

Below is a sample configuration that uses a RollingFileAppender with the cron triggering policy configured to trigger every day at midnight. Archives are stored in a directory based on the current
year and month. All files under the base directory that match the "*/app-*.log.gz" glob and are 60
days old or older are deleted at rollover time.

```xml
<Configuration status="warn" name="MyApp" packages="">
  <Properties>
    <Property name="baseDir">logs</Property>
  </Properties>
  <Appenders>
    <RollingFile name="RollingFile" fileName="${baseDir}/app.log"
      filePattern="${baseDir}/${date:yyyy-MM}/app-%d{yyyy-MM-dd}.log.gz">
      <PatternLayout pattern="%d %p %c{1.} [%t] %m%n" />
      <CronTriggeringPolicy schedule="0 0 0 * * ?"/>
      <DefaultRolloverStrategy>
        <Delete basePath="${baseDir}" maxDepth="2">
          <IfFileName glob="*/app-*.log.gz" />
          <IfLastModified age="60d" />
        </Delete>
      </DefaultRolloverStrategy>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

Below is a sample configuration that uses a RollingFileAppender with both the time and size based
triggering policies, will create up to 100 archives on the same day (1-100) that are stored in a
directory based on the current year and month, and will compress each archive using gzip and will
roll every hour. During every rollover, this configuration will delete files that match "*/app-*.log.gz"
and are 30 days old or older, but keep the most recent 100 GB or the most recent 10 files, whichever
comes first.
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Properties>
    <Property name="baseDir">logs</Property>
  </Properties>
  <Appenders>
    <RollingFile name="RollingFile" fileName="${baseDir}/app.log"
      filePattern="${baseDir}/${date:yyyy-MM}/app-%d{yyyy-MM-dd-HH}-%i.log.gz">
      <PatternLayout pattern="%d %p %c{1.} [%t] %m%n" />
      <Policies>
        <TimeBasedTriggeringPolicy />
        <SizeBasedTriggeringPolicy size="250 MB"/>
      </Policies>
      <DefaultRolloverStrategy max="100">
        <!--
        Nested conditions: the inner condition is only evaluated on files
        for which the outer conditions are true.
        -->
        <Delete basePath="${baseDir}" maxDepth="2">
          <IfFileName glob="*/app-*.log.gz">
            <IfLastModified age="30d">
              <IfAny>
                <IfAccumulatedFileSize exceeds="100 GB"/>
                <IfAccumulatedFileCount exceeds="10"/>
              </IfAny>
            </IfLastModified>
          </IfFileName>
        </Delete>
      </DefaultRolloverStrategy>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>script</td>
<td>Script, ScriptFile or ScriptRef</td>
<td>The Script element that specifies the logic to be executed. The script is passed a list of paths found under the base path and must return the paths to delete as a java.util.List&lt;java.io.File&gt; See also the ScriptFilter documentation for an example of how ScriptFiles and ScriptRefs can be configured.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>basePath</td>
<td>java.nio.file.Path</td>
<td>The directory from where the Delete action started scanning for files to delete. Can be used to relativize the paths in the pathList.</td>
</tr>
<tr>
<td>pathList</td>
<td>java.util.List&lt;PathWithAttributes&gt;</td>
<td>The list of paths found under the base path up to the specified max depth, sorted most recently modified files first. The script is free to modify and return this list.</td>
</tr>
<tr>
<td>statusLogger</td>
<td>StatusLogger</td>
<td>The StatusLogger that can be used to log internal events during script execution.</td>
</tr>
<tr>
<td>configuration</td>
<td>Configuration</td>
<td>The Configuration that owns this ScriptCondition.</td>
</tr>
<tr>
<td>substitutor</td>
<td>StrSubstitutor</td>
<td>The StrSubstitutor used to replace lookup variables.</td>
</tr>
<tr>
<td>?</td>
<td>String</td>
<td>Any properties declared in the configuration.</td>
</tr>
</tbody>
</table>

Script Parameters

Below is a sample configuration that uses a RollingFileAppender with the cron triggering policy configured to trigger every day at midnight. Archives are stored in a directory based on the current year and month. The script returns a list of rolled over files under the base directory dated Friday the 13th. The Delete action will delete all files returned by the script.
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="trace" name="MyApp" packages="">
  <Properties>
    <Property name="baseDir">logs</Property>
  </Properties>
  <Appenders>
    <RollingFile name="RollingFile" fileName="${baseDir}/app.log" filePattern="${baseDir}/${date:yyyy-MM}/app-%d{yyyyMMdd}.log.gz">
      <PatternLayout pattern="%d %p %c{1.} [%t] %m%n" />
      <CronTriggeringPolicy schedule="0 0 0 * * ?/>
      <DefaultRolloverStrategy>
        <Delete basePath="${baseDir}" maxDepth="2">
          <ScriptCondition>
            <Script name="superstitious" language="groovy">
              import java.nio.file.*
              def result = [];
              def pattern = "\d*\/app-(\d*)\.log\.gz/";
              pathList.each { pathWithAttributes ->
                def relative = basePath.relativize pathWithAttributes.path
                statusLogger.trace 'SCRIPT: relative path=' + relative + " (base="$basePath")";
                def matcher = pattern.matcher(relative.toString());
                if (matcher.find()) {
                  def dateString = matcher.group(1);
                  def calendar = Date.parse("yyyyMMdd", dateString).toCalendar();
                  def friday13th = calendar.get(Calendar.DAY_OF_MONTH) == 13 && calendar.get(Calendar.DAY_OF_WEEK) == Calendar.FRIDAY;
                  if (friday13th) {
                    result.add pathWithAttributes;
                    statusLogger.trace 'SCRIPT: deleting path ' + pathWithAttributes;
                  }
                }
              }
              statusLogger.trace 'SCRIPT: returning ' + result;
              result;
            }
          </ScriptCondition>
        </Delete>
        </DefaultRolloverStrategy>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
Log4j-2.9 introduces a `PosixViewAttribute` action that gives users more control over which file attribute permissions, owner and group should be applied. The `PosixViewAttribute` action lets users configure one or more conditions that select the eligible files relative to a base directory.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basePath</td>
<td>String</td>
<td>Required. Base path from where to start scanning for files to apply attributes.</td>
</tr>
<tr>
<td>maxDepth</td>
<td>int</td>
<td>The maximum number of levels of directories to visit. A value of 0 means that only the starting file (the base path itself) is visited, unless denied by the security manager. A value of Integer.MAX_VALUE indicates that all levels should be visited. The default is 1, meaning only the files in the specified base directory.</td>
</tr>
<tr>
<td>followLinks</td>
<td>boolean</td>
<td>Whether to follow symbolic links. Default is false.</td>
</tr>
<tr>
<td>pathConditions</td>
<td>PathCondition[]</td>
<td>see <a href="#">DeletePathCondition</a></td>
</tr>
<tr>
<td>filePermissions</td>
<td>String</td>
<td>File attribute permissions in POSIX format to apply when action is executed. Underlying files system shall support POSIX file attribute view. Examples: rw------ or rw-rw-rw- etc...</td>
</tr>
<tr>
<td>fileOwner</td>
<td>String</td>
<td>File owner to define when action is executed. Changing file's owner may be restricted for security reason and Operation not permitted IOException thrown. Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file if <code>_POSIX_CHOWN_RESTRICTED</code> is in effect for path. Underlying files system shall support file owner attribute view.</td>
</tr>
</tbody>
</table>
fileGroup String File group to define when an action is executed. Underlying files system shall support POSIX file attribute view.

PosixViewAttribute Parameters

Below is a sample configuration that uses a RollingFileAppender and defines different POSIX file attribute view for current and rolled log files.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="trace" name="MyApp" packages="">
  <Properties>
    <Property name="baseDir">logs</Property>
  </Properties>
  <Appenders>
    <RollingFile name="RollingFile" fileName="${baseDir}/app.log" filePattern="${baseDir}/${date:yyyy-MM}/app-%d{yyyyMMdd}.log.gz" filePermissions="rw-------">
      <PatternLayout pattern="%d %p %c{1.} [%t] %m%n"/>
      <CronTriggeringPolicy schedule="0 0 0 * * ?"/>
      <DefaultRolloverStrategy stopCustomActionsOnError="true">
        <PosixViewAttribute basePath="${baseDir}/${date:yyyy-MM}" filePermissions="r--r--r--">
          <IfFileName glob="*.gz"/>
        </PosixViewAttribute>
      </DefaultRolloverStrategy>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.22 RollingRandomAccessFileAppender

The RollingRandomAccessFileAppender is similar to the standard RollingFileAppender except it is always buffered (this cannot be switched off) and internally it uses a ByteBuffer + RandomAccessFile instead of a BufferedOutputStream. We saw a 20-200% performance improvement compared to RollingFileAppender with "bufferedIO=true" in our measurements. The RollingRandomAccessFileAppender writes to the File named in the fileName parameter and rolls the file over according the TriggeringPolicy and the RolloverPolicy. Similar to the RollingFileAppender, RollingRandomAccessFileAppender uses a RollingRandomAccessFileManager to actually perform the file I/O and perform the rollover. While RollingRandomAccessFileAppender from different Configurations cannot be shared, the RollingRandomAccessFileManagers can be if the Manager is accessible. For example, two web applications in a servlet container can have their own configuration and safely write to the same file if Log4j is in a ClassLoader that is common to both of them.
A RollingRandomAccessFileAppender requires a `TriggeringPolicy` and a `RolloverStrategy`. The triggering policy determines if a rollover should be performed while the RolloverStrategy defines how the rollover should be done. If no RolloverStrategy is configured, RollingRandomAccessFileAppender will use the `DefaultRolloverStrategy`. Since log4j-2.5, a custom delete action can be configured in the DefaultRolloverStrategy to run at rollover.

File locking is not supported by the RollingRandomAccessFileAppender.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>boolean</td>
<td>When true - the default, records will be appended to the end of the file. When set to false, the file will be cleared before new records are written.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.</td>
</tr>
<tr>
<td>filePattern</td>
<td>String</td>
<td>The pattern of the file name of the archived log file. The format of the pattern should be dependent on the RolloverPolicy that is used. The DefaultRolloverPolicy will accept both a date/time pattern compatible with SimpleDateFormat and/or a %i which represents an integer counter. The pattern also supports interpolation at runtime so any of the Lookups (such as the DateLookup) can be included in the pattern.</td>
</tr>
<tr>
<td>immediateFlush</td>
<td>boolean</td>
<td>When set to true - the default, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance.</td>
</tr>
</tbody>
</table>

Flushing after every write is only useful when using this appender with synchronous loggers. Asynchronous loggers and appenders will automatically flush at the end of a batch of events, even if immediateFlush is set to false. This also guarantees the data is written to disk but is more efficient.
| **bufferSize** | int  | The buffer size, defaults to 262,144 bytes (256 * 1024). |
| **layout**      | Layout | The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of "%m%n" will be used. |
| **name**        | String | The name of the Appender. |
| **policy**      | TriggeringPolicy | The policy to use to determine if a rollover should occur. |
| **strategy**    | RolloverStrategy | The strategy to use to determine the name and location of the archive file. |
| **ignoreExceptions** | boolean | The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender. |
| **filePermissions** | String | File attribute permissions in POSIX format to apply whenever the file is created. Underlying files system shall support POSIX file attribute view. Examples: rw-------- or rw-rw-rw- etc... |
| **fileOwner**   | String | File owner to define whenever the file is created. Changing file’s owner may be restricted for security reason and Operation not permitted IOException thrown. Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file if _POSIX_CHOWN_RESTRICTED is in effect for path. Underlying files system shall support file owner attribute view. |
RollingRandomAccessFileAppender Parameters

9.1.22.1 Triggering Policies
See RollingFileAppender Triggering Policies.

9.1.22.2 Rollover Strategies
See RollingFileAppender Rollover Strategies.

Below is a sample configuration that uses a RollingRandomAccessFileAppender with both the time and size based triggering policies, will create up to 7 archives on the same day (1-7) that are stored in a directory based on the current year and month, and will compress each archive using gzip:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
    <Appenders>
        <RollingRandomAccessFile name="RollingRandomAccessFile" fileName="logs/app.log" filePattern="logs/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
            </PatternLayout>
            <Policies>
                <TimeBasedTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="250 MB" />
            </Policies>
        </RollingRandomAccessFile>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="RollingRandomAccessFile" />
        </Root>
    </Loggers>
</Configuration>
```

This second example shows a rollover strategy that will keep up to 20 files before removing them.
Below is a sample configuration that uses a RollingRandomAccessFileAppender with both the time and size based triggering policies, will create up to 7 archives on the same day (1-7) that are stored in a directory based on the current year and month, and will compress each archive using gzip and will roll every 6 hours when the hour is divisible by 6:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingRandomAccessFile name="RollingRandomAccessFile" fileName="logs/app.log" filePattern="logs/$${date:yyyy-MM}/app-%d{yyyy-MM-dd-HH}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <TimeBasedTriggeringPolicy interval="6" modulate="true"/>
        <SizeBasedTriggeringPolicy size="250 MB"/>
      </Policies>
    </RollingRandomAccessFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingRandomAccessFile"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.23 RoutingAppender

The RoutingAppender evaluates LogEvents and then routes them to a subordinate Appender. The target Appender may be an appender previously configured and may be referenced by its name or the Appender can be dynamically created as needed. The RoutingAppender should be configured after any Appenders it references to allow it to shut down properly.
You can also configure a RoutingAppender with scripts: you can run a script when the appender starts and when a route is chosen for a log event.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>RewritePolicy</td>
<td>RewritePolicy</td>
<td>The RewritePolicy that will manipulate the LogEvent.</td>
</tr>
<tr>
<td>Routes</td>
<td>Routes</td>
<td>Contains one or more Route declarations to identify the criteria for choosing Appenders.</td>
</tr>
<tr>
<td>Script</td>
<td>Script</td>
<td>This Script runs when Log4j starts the RoutingAppender and returns a String Route key to determine the default Route. This script is passed the following variables:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration</td>
<td>Configuration</td>
<td>The active Configuration.</td>
</tr>
<tr>
<td>staticVariables</td>
<td>Map</td>
<td>A Map shared between all script invocations for this appender instance. This is the same map passed to the Routes Script.</td>
</tr>
</tbody>
</table>

RoutingAppender Script Parameters
RoutingAppender Parameters

In this example, the script causes the "ServiceWindows" route to be the default route on Windows and "ServiceOther" on all other operating systems. Note that the List Appender is one of our test appenders, any appender can be used, it is only used as a shorthand.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN" name="RoutingTest">
    <Appenders>
        <Routing name="Routing">
            <Script name="RoutingInit" language="JavaScript"><![CDATA[
                importPackage(java.lang);
                System.getProperty("os.name").search("Windows") > -1 ? "ServiceWindows" : "ServiceOther";]]>
            </Script>
            <Routes>
                <Route key="ServiceOther">
                    <List name="List1" />
                </Route>
                <Route key="ServiceWindows">
                    <List name="List2" />
                </Route>
            </Routes>
        </Routing>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="Routing" />
        </Root>
    </Loggers>
</Configuration>
```

9.1.23.1 Routes

The Routes element accepts a single attribute named "pattern". The pattern is evaluated against all the registered Lookups and the result is used to select a Route. Each Route may be configured with a key. If the key matches the result of evaluating the pattern then that Route will be selected. If no key is specified on a Route then that Route is the default. Only one Route can be configured as the default.

The Routes element may contain a Script child element. If specified, the Script is run for each log event and returns the String Route key to use.

You must specify either the pattern attribute or the Script element, but not both.

Each Route must reference an Appender. If the Route contains a ref attribute then the Route will reference an Appender that was defined in the configuration. If the Route contains an Appender definition then an Appender will be created within the context of the RoutingAppender and will be reused each time a matching Appender name is referenced through a Route.
This script is passed the following variables:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration</td>
<td>Configuration</td>
<td>The active Configuration.</td>
</tr>
<tr>
<td>staticVariables</td>
<td>Map</td>
<td>A Map shared between all script invocations for this appender instance. This is the same map passed to the Routes Script.</td>
</tr>
<tr>
<td>logEvent</td>
<td>LogEvent</td>
<td>The log event.</td>
</tr>
</tbody>
</table>

RoutingAppender Routes Script Parameters

In this example, the script runs for each log event and picks a route based on the presence of a Marker named "AUDIT".
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN" name="RoutingTest">
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT" />
    <Flume name="AuditLogger" compress="true">
      <Agent host="192.168.10.101" port="8800"/>
      <Agent host="192.168.10.102" port="8800"/>
      <RFC5424Layout enterpriseNumber="18060" includeMDC="true" appName="MyApp"/>
    </Flume>
    <Routing name="Routing">
      <Routes>
        <Script name="RoutingInit" language="JavaScript">
          <![CDATA[
            if (logEvent.getMarker() != null && logEvent.getMarker().isInstanceOf("AUDIT")) {
              return "AUDIT";
            } else if (logEvent.getContextMap().containsKey("UserId")) {
              return logEvent.getContextMap().get("UserId");
            }
            return "STDOUT";]]>
        </Script>
        <Route>
          <RollingFile
            name="Rolling-${mdc:UserId}" fileName="${mdc:UserId}.log" filePattern="${mdc:UserId}.%i.log.gz">
            <PatternLayout>
              <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <SizeBasedTriggeringPolicy size="500"/>
          </RollingFile>
        </Route>
        <Route ref="AuditLogger" key="AUDIT"/>
        <Route ref="STDOUT" key="STDOUT"/>
      </Routes>
      <IdlePurgePolicy timeToLive="15" timeUnit="minutes"/>
    </Routing>
    <Loggers>
      <Root level="error">
        <AppenderRef ref="Routing" />
      </Root>
    </Loggers>
  </Appenders>
</Configuration>

### 9.1.23.2 Purge Policy

The RoutingAppender can be configured with a PurgePolicy whose purpose is to stop and remove dormant Appenders that have been dynamically created by the RoutingAppender. Log4j currently provides the IdlePurgePolicy as the only PurgePolicy available for cleaning up the Appenders. The IdlePurgePolicy accepts 2 attributes; timeToLive, which is the number of timeUnits the Appender should survive without having any events sent to it, and timeUnit, the String representation of java.util.concurrent.TimeUnit which is used with the timeToLive attribute.
Below is a sample configuration that uses a RoutingAppender to route all Audit events to a FlumeAppender and all other events will be routed to a RollingFileAppender that captures only the specific event type. Note that the AuditAppender was predefined while the RollingFileAppenders are created as needed.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Flume name="AuditLogger" compress="true">
      <Agent host="192.168.10.101" port="8800"/>
      <Agent host="192.168.10.102" port="8800"/>
      <RFC5424Layout enterpriseNumber="18060" includeMDC="true" appName="MyApp"/>
    </Flume>
    <Routing name="Routing">
      <Routes pattern="$$sd:type"">
        <Route>
          <RollingFile name="Rolling-${sd:type}" fileName="${sd:type}.log" filePattern="${sd:type}.%i.log.gz">
            <PatternLayout>
              <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <SizeBasedTriggeringPolicy size="500"/>
          </RollingFile>
        </Route>
        <Route ref="AuditLogger" key="Audit"/>
      </Routes>
    </Routing>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Routing"/>
    </Root>
  </Loggers>
</Configuration>

9.1.24 SMTPAppender

Sends an e-mail when a specific logging event occurs, typically on errors or fatal errors.

The number of logging events delivered in this e-mail depend on the value of BufferSize option. The SMTPAppender keeps only the last BufferSize logging events in its cyclic buffer. This keeps memory requirements at a reasonable level while still delivering useful application context. All events in the buffer are included in the email. The buffer will contain the most recent events of level TRACE to WARN preceding the event that triggered the email.

The default behavior is to trigger sending an email whenever an ERROR or higher severity event is logged and to format it as HTML. The circumstances on when the email is sent can be controlled by setting one or more filters on the Appender. As with other Appenders, the formatting can be controlled by specifying a Layout for the Appender.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>from</td>
<td>String</td>
<td>The email address of the sender.</td>
</tr>
<tr>
<td>replyTo</td>
<td>String</td>
<td>The comma-separated list of reply-to email addresses.</td>
</tr>
<tr>
<td>to</td>
<td>String</td>
<td>The comma-separated list of recipient email addresses.</td>
</tr>
<tr>
<td>cc</td>
<td>String</td>
<td>The comma-separated list of CC email addresses.</td>
</tr>
<tr>
<td>bcc</td>
<td>String</td>
<td>The comma-separated list of BCC email addresses.</td>
</tr>
<tr>
<td>subject</td>
<td>String</td>
<td>The subject of the email message.</td>
</tr>
<tr>
<td>bufferSize</td>
<td>integer</td>
<td>The maximum number of log events to be buffered for inclusion in the message. Defaults to 512.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. If no layout is supplied HTML layout will be used.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>smtpDebug</td>
<td>boolean</td>
<td>When set to true enables session debugging on STDOUT. Defaults to false.</td>
</tr>
<tr>
<td>smtpHost</td>
<td>String</td>
<td>The SMTP hostname to send to.</td>
</tr>
<tr>
<td>smtpPassword</td>
<td>String</td>
<td>The password required to authenticate against the SMTP server.</td>
</tr>
<tr>
<td>smtpPort</td>
<td>integer</td>
<td>The SMTP port to send to.</td>
</tr>
<tr>
<td>smtpProtocol</td>
<td>String</td>
<td>The SMTP transport protocol (such as &quot;smtps&quot;, defaults to &quot;smtp&quot;).</td>
</tr>
<tr>
<td>smtpUsername</td>
<td>String</td>
<td>The username required to authenticate against the SMTP server.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
</tbody>
</table>

SMTPAppender Parameters
9.1.25 ScriptAppenderSelector

When the configuration is built, the ScriptAppenderSelector appender calls a Script to compute an appender name. Log4j then creates one of the appender named listed under AppenderSet using the name of the ScriptAppenderSelector. After configuration, Log4j ignores the ScriptAppenderSelector. Log4j only builds the one selected appender from the configuration tree, and ignores other AppenderSet child nodes.

In the following example, the script returns the name "List2". The appender name is recorded under the name of the ScriptAppenderSelector, not the name of the selected appender, in this example, "SelectIt".

```xml
<Configuration status="WARN" name="ScriptAppenderSelectorExample">
  <Appenders>
    <ScriptAppenderSelector name="SelectIt">
      <Script language="JavaScript">
        <![CDATA[
          importPackage(java.lang);
          System.getProperty("os.name").search("Windows") > -1 ? "MyCustomWindowsAppender" : "MySyslogAppender";
        ]]>
      </Script>
      <AppenderSet>
        <MyCustomWindowsAppender name="MyAppender" ... />
        <SyslogAppender name="MySyslog" ... />
      </AppenderSet>
    </ScriptAppenderSelector>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="SelectIt"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.26 SocketAppender

The SocketAppender is an OutputStreamAppender that writes its output to a remote destination specified by a host and port. The data can be sent over either TCP or UDP and can be sent in any format. You can optionally secure communication with SSL.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender.</td>
</tr>
<tr>
<td>host</td>
<td>String</td>
<td>The name or address of the system that is listening for log events. This parameter is required.</td>
</tr>
<tr>
<td>port</td>
<td>integer</td>
<td>The port on the host that is listening for log events. This parameter must be specified.</td>
</tr>
<tr>
<td>protocol</td>
<td>String</td>
<td>&quot;TCP&quot; (default), &quot;SSL&quot; or &quot;UDP&quot;.</td>
</tr>
<tr>
<td>SSL</td>
<td>SslConfiguration</td>
<td>Contains the configuration for the KeyStore and TrustStore. See SSL.</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>immediateFail</td>
<td>boolean</td>
<td>When set to true, log events will not wait to try to reconnect and will fail immediately if the socket is not available.</td>
</tr>
<tr>
<td>immediateFlush</td>
<td>boolean</td>
<td>When set to true - the default, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance.</td>
</tr>
<tr>
<td>bufferedIO</td>
<td>boolean</td>
<td>When true - the default, events are written to a buffer and the data will be written to the socket when the buffer is full or, if immediateFlush is set, when the record is written.</td>
</tr>
<tr>
<td>bufferSize</td>
<td>int</td>
<td>When bufferedIO is true, this is the buffer size, the default is 8192 bytes.</td>
</tr>
<tr>
<td>layout</td>
<td>Layout</td>
<td>The Layout to use to format the LogEvent. Required, there is no default. New since 2.9, in previous versions SerializedLayout was default.</td>
</tr>
<tr>
<td>reconnectionDelayMillis</td>
<td>integer</td>
<td>If set to a value greater than 0, after an error the SocketManager will attempt to reconnect to the server after waiting the specified number of milliseconds. If the reconnect fails then an exception will be thrown (which can be caught by the application if ignoreExceptions is set to false).</td>
</tr>
</tbody>
</table>
SocketAppender Parameters

This is an unsecured TCP configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
<Appenders>
    <Socket name="socket" host="localhost" port="9500">
        <JsonLayout properties="true"/>
    </Socket>
</Appenders>
<Loggers>
    <Root level="error">
        <AppenderRef ref="socket"/>
    </Root>
</Loggers>
</Configuration>
```

This is a secured SSL configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
<Appenders>
    <Socket name="socket" host="localhost" port="9500">
        <JsonLayout properties="true"/>
        <SSL>
            <KeyStore location="log4j2-keystore.jks" passwordEnvironmentVariable="KEYSTORE_PASSWORD"/>
            <TrustStore location="truststore.jks" passwordFile="${sys:user.home}/truststore.pwd"/>
        </SSL>
    </Socket>
</Appenders>
<Loggers>
    <Root level="error">
        <AppenderRef ref="socket"/>
    </Root>
</Loggers>
</Configuration>
```
9.1.27 SSL Configuration

Several appenders can be configured to use either a plain network connection or a Secure Socket Layer (SSL) connection. This section documents the parameters available for SSL configuration.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>String</td>
<td>SSL if omitted. See also Standard names.</td>
</tr>
<tr>
<td>KeyStore</td>
<td>KeyStore</td>
<td>Contains your private keys and certificates, and determines which authentication credentials to send to the remote host.</td>
</tr>
<tr>
<td>TrustStore</td>
<td>TrustStore</td>
<td>Contains the CA certificates of the remote counterparty. Determines whether the remote authentication credentials (and thus the connection) should be trusted.</td>
</tr>
</tbody>
</table>

SSL Configuration Parameters

9.1.27.1 KeyStore

The keystore is meant to contain your private keys and certificates, and determines which authentication credentials to send to the remote host.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>String</td>
<td>Path to the keystore file.</td>
</tr>
<tr>
<td>password</td>
<td>char[]</td>
<td>Plain text password to access the keystore. Cannot be combined with either passwordEnvironmentVariable or passwordFile.</td>
</tr>
<tr>
<td>passwordEnvironmentVariable</td>
<td>String</td>
<td>Name of an environment variable that holds the password. Cannot be combined with either password or passwordFile.</td>
</tr>
<tr>
<td>passwordFile</td>
<td>String</td>
<td>Path to a file that holds the password. Cannot be combined with either password or passwordEnvironmentVariable.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Optional KeyStore type, e.g., JKS, PKCS12, PKCS11, BKS, Windows–MY/Windows–ROOT, KeychainStore, etc. The default is JKS. See also Standard types.</td>
</tr>
<tr>
<td>keyManagerFactoryAlgorithm</td>
<td>String</td>
<td>Optional KeyManagerFactory algorithm. The default is SunX509. See also Standard algorithms.</td>
</tr>
</tbody>
</table>
9.1.27.2 TrustStore

The trust store is meant to contain the CA certificates you are willing to trust when a remote party presents its certificate. Determines whether the remote authentication credentials (and thus the connection) should be trusted.

In some cases, they can be one and the same store, although it is often better practice to use distinct stores (especially when they are file-based).

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>String</td>
<td>Path to the keystore file.</td>
</tr>
<tr>
<td>password</td>
<td>char[]</td>
<td>Plain text password to access the keystore. Cannot be combined with either passwordEnvironmentVariable or passwordFile.</td>
</tr>
<tr>
<td>passwordEnvironmentVariable</td>
<td>String</td>
<td>Name of an environment variable that holds the password. Cannot be combined with either password or passwordFile.</td>
</tr>
<tr>
<td>passwordFile</td>
<td>String</td>
<td>Path to a file that holds the password. Cannot be combined with either password or passwordEnvironmentVariable.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Optional KeyStore type, e.g. JKS, PKCS12, PKCS11, BKS, Windows-MY/Windows-ROOT, KeychainStore, etc. The default is JKS. See also Standard types.</td>
</tr>
<tr>
<td>trustManagerFactoryAlgorithm</td>
<td>String</td>
<td>Optional TrustManagerFactory algorithm. The default is SunX509. See also Standard algorithms.</td>
</tr>
</tbody>
</table>

TrustStore Configuration Parameters

9.1.27.3 Example

```xml
...<SSL>
  <KeyStore location="log4j2-keystore.jks" passwordEnvironmentVariable="KEYSTORE_PASSWORD"/>
  <TrustStore location="truststore.jks" passwordFile="$sys:user.home]/truststore.pwd"/>
</SSL>
...
```

9.1.28 SyslogAppender

The SyslogAppender is a SocketAppender that writes its output to a remote destination specified by a host and port in a format that conforms with either the BSD Syslog format or the RFC 5424 format. The data can be sent over either TCP or UDP.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>advertise</td>
<td>boolean</td>
<td>Indicates whether the appender should be advertised.</td>
</tr>
<tr>
<td>appName</td>
<td>String</td>
<td>The value to use as the APP-NAME in the RFC 5424 syslog record.</td>
</tr>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting the syslog String to a byte array. The String must be a valid Charset. If not specified, the default system Charset will be used.</td>
</tr>
<tr>
<td>connectTimeoutMillis</td>
<td>integer</td>
<td>The connect timeout in milliseconds. The default is 0 (infinite timeout, like Socket.connect() methods).</td>
</tr>
<tr>
<td>enterpriseNumber</td>
<td>integer</td>
<td>The IANA enterprise number as described in RFC 5424</td>
</tr>
<tr>
<td>filter</td>
<td>Filter</td>
<td>A Filter to determine if the event should be handled by this Appender. More than one Filter may be used by using a CompositeFilter.</td>
</tr>
<tr>
<td>facility</td>
<td>String</td>
<td>The facility is used to try to classify the message. The facility option must be set to one of &quot;KERN&quot;, &quot;USER&quot;, &quot;MAIL&quot;, &quot;DAEMON&quot;, &quot;AUTH&quot;, &quot;SYSLOG&quot;, &quot;LPR&quot;, &quot;NEWS&quot;, &quot;UUCP&quot;, &quot;CRON&quot;, &quot;AUTHPRIV&quot;, &quot;FTP&quot;, &quot;NTP&quot;, &quot;AUDIT&quot;, &quot;ALERT&quot;, &quot;CLOCK&quot;, &quot;LOCAL0&quot;, &quot;LOCAL1&quot;, &quot;LOCAL2&quot;, &quot;LOCAL3&quot;, &quot;LOCAL4&quot;, &quot;LOCAL5&quot;, &quot;LOCAL6&quot;, or &quot;LOCAL7&quot;. These values may be specified as upper or lower case characters.</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>If set to &quot;RFC5424&quot; the data will be formatted in accordance with RFC 5424. Otherwise, it will be formatted as a BSD Syslog record. Note that although BSD Syslog records are required to be 1024 bytes or shorter the SyslogLayout does not truncate them. The RFC5424Layout also does not truncate records since the receiver must accept records of up to 2048 bytes and may accept records that are longer.</td>
</tr>
<tr>
<td>host</td>
<td>String</td>
<td>The name or address of the system that is listening for log events. This parameter is required.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>id</td>
<td>String</td>
<td>The default structured data id to use when formatting according to RFC 5424. If the LogEvent contains a StructuredDataMessage the id from the Message will be used instead of this value.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>The default is true, causing exceptions encountered while appending events to be internally logged and then ignored. When set to false exceptions will be propagated to the caller, instead. You must set this to false when wrapping this Appender in a FailoverAppender.</td>
</tr>
<tr>
<td>immediateFail</td>
<td>boolean</td>
<td>When set to true, log events will not wait to try to reconnect and will fail immediately if the socket is not available.</td>
</tr>
<tr>
<td>immediateFlush</td>
<td>boolean</td>
<td>When set to true - the default, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance.</td>
</tr>
<tr>
<td>includeMDC</td>
<td>boolean</td>
<td>Indicates whether data from the ThreadContextMap will be included in the RFC 5424 Syslog record. Defaults to true.</td>
</tr>
<tr>
<td>Layout</td>
<td>Layout</td>
<td>A custom layout which overrides the format setting.</td>
</tr>
<tr>
<td>loggerFields</td>
<td>List of KeyValuePairs</td>
<td>Allows arbitrary PatternLayout patterns to be included as specified ThreadContext fields; no default specified. To use, include a &gt;LoggerFields&lt; nested element, containing one or more &gt;KeyValuePair&lt; elements. Each &gt;KeyValuePair&lt; must have a key attribute, which specifies the key name which will be used to identify the field within the MDC Structured Data element, and a value attribute, which specifies the PatternLayout pattern to use as the value.</td>
</tr>
<tr>
<td>mdcExcludes</td>
<td>String</td>
<td>A comma separated list of mdc keys that should be excluded from the LogEvent. This is mutually exclusive with the mdcIncludes attribute. This attribute only applies to RFC 5424 syslog records.</td>
</tr>
</tbody>
</table>
SyslogAppender Parameters

A sample syslogAppender configuration that is configured with two SyslogAppenders, one using the BSD format and one using RFC 5424.
For **SSL** this appender writes its output to a remote destination specified by a host and port over SSL in a format that conforms with either the BSD Syslog format or the RFC 5424 format.

### 9.1.29 ZeroMQ/JeroMQ Appender

The ZeroMQ appender uses the **JeroMQ** library to send log events to one or more ZeroMQ endpoints.

This is a simple JeroMQ configuration:
<?xml version="1.0" encoding="UTF-8"?>
<Configuration name="JeroMQAppenderTest" status="TRACE">
  <Appenders>
    <JeroMQ name="JeroMQAppender">
      <Property name="endpoint">tcp://*:5556</Property>
      <Property name="endpoint">ipc://info-topic</Property>
    </JeroMQ>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="JeroMQAppender"/>
    </Root>
  </Loggers>
</Configuration>

The table below describes all options. Please consult the JeroMQ and ZeroMQ documentation for details.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the Appender. Required.</td>
</tr>
<tr>
<td>Layout</td>
<td>layout</td>
<td>The Layout to use to format the LogEvent. If no layout is supplied the default pattern layout of &quot;%m%n&quot; will be used.</td>
</tr>
<tr>
<td>Filters</td>
<td>Filter</td>
<td>The Filter(s) of the Appender.</td>
</tr>
<tr>
<td>Properties</td>
<td>Property[]</td>
<td>One or more Property elements, named endpoint.</td>
</tr>
<tr>
<td>ignoreExceptions</td>
<td>boolean</td>
<td>If true, exceptions will be logged and suppressed. If false errors will be logged and then passed to the application.</td>
</tr>
<tr>
<td>affinity</td>
<td>long</td>
<td>The ZMQ_AFFINITY option. Defaults to 0.</td>
</tr>
<tr>
<td>backlog</td>
<td>long</td>
<td>The ZMQ_BACKLOG option. Defaults to 100.</td>
</tr>
<tr>
<td>delayAttachOnConnect</td>
<td>boolean</td>
<td>The ZMQ_DELAY_ATTACHMENT_ON_CONNECT option. Defaults to false.</td>
</tr>
<tr>
<td>identity</td>
<td>byte[]</td>
<td>The ZMQ_IDENTITY option. Defaults to none.</td>
</tr>
<tr>
<td>ipv4Only</td>
<td>boolean</td>
<td>The ZMQ_IPV4ONLY option. Defaults to true.</td>
</tr>
<tr>
<td>linger</td>
<td>long</td>
<td>The ZMQ_LINGER option. Defaults to -1.</td>
</tr>
<tr>
<td>maxMsgSize</td>
<td>long</td>
<td>The ZMQ_MAXMSGSIZE option. Defaults to -1.</td>
</tr>
<tr>
<td>rcvHwm</td>
<td>long</td>
<td>The ZMQ_RCVHWM option. Defaults to 1000.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>receiveBufferSize</td>
<td>long</td>
<td>The ZMQ_RCVBUF option. Defaults to 0.</td>
</tr>
<tr>
<td>receiveTimeOut</td>
<td>int</td>
<td>The ZMQ_RCVTIMEO option. Defaults to -1.</td>
</tr>
<tr>
<td>reconnectIVL</td>
<td>long</td>
<td>The ZMQ_RECONNECT_IVL option. Defaults to 100.</td>
</tr>
<tr>
<td>reconnectIVLMax</td>
<td>long</td>
<td>The ZMQ_RECONNECT_IVL_MAX option. Defaults to 0.</td>
</tr>
<tr>
<td>sendBufferSize</td>
<td>long</td>
<td>The ZMQ_SNDBUF option. Defaults to 0.</td>
</tr>
<tr>
<td>sendTimeOut</td>
<td>int</td>
<td>The ZMQ_SNDTIMEO option. Defaults to -1.</td>
</tr>
<tr>
<td>sndHwm</td>
<td>long</td>
<td>The ZMQ_SNDHWM option. Defaults to 1000.</td>
</tr>
<tr>
<td>tcpKeepAlive</td>
<td>int</td>
<td>The ZMQ_TCP_KEEPALIVE option. Defaults to -1.</td>
</tr>
<tr>
<td>tcpKeepAliveCount</td>
<td>long</td>
<td>The ZMQ_TCP_KEEPALIVE_CNT option. Defaults to -1.</td>
</tr>
<tr>
<td>tcpKeepAliveIdle</td>
<td>long</td>
<td>The ZMQ_TCP_KEEPALIVE_IDLE option. Defaults to -1.</td>
</tr>
<tr>
<td>tcpKeepAliveInterval</td>
<td>long</td>
<td>The ZMQ_TCP_KEEPALIVE_INTVL option. Defaults to -1.</td>
</tr>
<tr>
<td>xpubVerbose</td>
<td>boolean</td>
<td>The ZMQ_XPUB_VERBOSE option. Defaults to false.</td>
</tr>
</tbody>
</table>

**JeroMQ Parameters**
10 Layouts

10.1 Layouts

An Appender uses a Layout to format a LogEvent into a form that meets the needs of whatever will be consuming the log event. In Log4j 1.x and Logback Layouts were expected to transform an event into a String. In Log4j 2 Layouts return a byte array. This allows the result of the Layout to be useful in many more types of Appenders. However, this means you need to configure most Layouts with a Charset to ensure the byte array contains correct values.

The root class for layouts that use a Charset is org.apache.logging.log4j.core.layout.AbstractStringLayout where the default is UTF-8. Each layout that extends AbstractStringLayout can provide its own default. See each layout below.

A custom character encoder was added to Log4j 2.4.1 for the ISO-8859-1 and US-ASCII charsets, to bring some of the performance improvements built-in to Java 8 to Log4j for use on Java 7. For applications that log only ISO-8859-1 characters, specifying this charset will improve performance significantly.

10.1.1 CSV Layouts

This layout creates Comma Separated Value (CSV) records and requires Apache Commons CSV 1.4.

The CSV layout can be used in two ways: First, using CsvParameterLayout to log event parameters to create a custom database, usually to a logger and file appender uniquely configured for this purpose. Second, using CsvLogEventLayout to log events to create a database, as an alternative to using a full DBMS or using a JDBC driver that supports the CSV format.

The CsvParameterLayout converts an event's parameters into a CSV record, ignoring the message. To log CSV records, you can use the usual Logger methods info(), debug(), and so on:

```
logger.info("Ignored", value1, value2, value3);
```

Which will create the CSV record:

```
value1, value2, value3
```

Alternatively, you can use a ObjectArrayMessage, which only carries parameters:

```
logger.info(new ObjectArrayMessage(value1, value2, value3));
```

The layouts CsvParameterLayout and CsvLogEventLayout are configured with the following parameters:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CsvParameterLayout and CsvLogEventLayout

Logging as a CSV events looks like this:

```java
logger.debug("one={}, two={}, three={}", 1, 2, 3);
```

Produces a CSV record with the following fields:

1. Time Nanos
2. Time Millis
3. Level
4. Thread ID
5. Thread Name
6. Thread Priority
7. Formatted Message
8. Logger FQCN
9. Logger Name
10. Marker
11. Thrown Proxy
12. Source
13. Context Map
14. Context Stack
Additional runtime dependencies are required for using CSV layouts.

### 10.1.2 GELF Layout

Lays out events in the Graylog Extended Log Format (GELF) 1.1. This layout compresses JSON to GZIP or ZLIB (the `compressionType`) if log event data is larger than 1024 bytes (the `compressionThreshold`). This layout does not implement chunking.

Configure as follows to send to a Graylog 2.x server with UDP:

```xml
<Appenders>
  <Socket name="Graylog" protocol="udp" host="graylog.domain.com" port="12201">
    <GelfLayout host="someserver" compressionType="ZLIB" compressionThreshold="1024"/>
  </Socket>
</Appenders>
```

Configure as follows to send to a Graylog 2.x server with TCP:

```xml
<Appenders>
  <Socket name="Graylog" protocol="tcp" host="graylog.domain.com" port="12201">
    <GelfLayout host="someserver" compressionType="OFF" includeNullDelimiter="true"/>
  </Socket>
</Appenders>
```

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>String</td>
<td>The value of the <code>host</code> property (optional, defaults to local host name).</td>
</tr>
<tr>
<td>compressionType</td>
<td>GZIP, ZLIB or OFF</td>
<td>Compression to use (optional, defaults to GZIP)</td>
</tr>
<tr>
<td>compressionThreshold</td>
<td>int</td>
<td>Compress if data is larger than this number of bytes (optional, defaults to 1024)</td>
</tr>
<tr>
<td>includeStacktrace</td>
<td>boolean</td>
<td>Whether to include full stacktrace of logged Throwables (optional, default to true). If set to false, only the class name and message of the Throwable will be included.</td>
</tr>
<tr>
<td>includeThreadContext</td>
<td>boolean</td>
<td>Whether to include thread context as additional fields (optional, default to true).</td>
</tr>
</tbody>
</table>
includeNullDelimiter

**GelfLayout Parameters**

To include any custom field in the output, use the following syntax:

```xml
<GelfLayout>
  <KeyValuePair key="additionalField1" value="constant value"/>
  <KeyValuePair key="additionalField2" value="${ctx:key}"/>
</GelfLayout>
```

Custom fields are included in the order they are declared. The values support lookups. See also:

- The [GELF specification](#)

### 10.1.3 HTML Layout

The HtmlLayout generates an HTML page and adds each LogEvent to a row in a table.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting the HTML String to a byte array. The value must be a valid <a href="#">Charset</a>. If not specified, this layout uses UTF-8.</td>
</tr>
<tr>
<td>contentType</td>
<td>String</td>
<td>The value to assign to the Content-Type header. The default is &quot;text/html&quot;.</td>
</tr>
<tr>
<td>locationInfo</td>
<td>boolean</td>
<td>If true, the filename and line number will be included in the HTML output. The default value is false. Generating location information is an expensive operation and may impact performance. Use with caution.</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>A String that will appear as the HTML title.</td>
</tr>
<tr>
<td>fontName</td>
<td>String</td>
<td>The font-family to use. The default is &quot;arial,sans-serif&quot;.</td>
</tr>
<tr>
<td>fontSize</td>
<td>String</td>
<td>The font-size to use. The default is &quot;small&quot;.</td>
</tr>
</tbody>
</table>
10.1.4 JSON Layout
Appends a series of JSON events as strings serialized as bytes.

10.1.4.1 Complete well-formed JSON vs. fragment JSON
If you configure complete="true", the appender outputs a well-formed JSON document. By default, with complete="false", you should include the output as an external file in a separate file to form a well-formed JSON document.

If complete="false", the appender does not write the JSON open array character "[" at the start of the document, "]" and the end, nor comma "," between records.

Log event follows this pattern:
If complete="false", the appender does not write the JSON open array character "[" at the start of the document, "]" and the end, nor comma "," between records.
10.1.4.2 Pretty vs. compact JSON

By default, the JSON layout is not compact (a.k.a. not "pretty") with `compact="false"`, which means the appender uses end-of-line characters and indents lines to format the text. If `compact="true"`, then no end-of-line or indentation is used. Message content may contain, of course, escaped end-of-lines.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting to a byte array. The value must be a valid Charset. If not specified, UTF-8 will be used.</td>
</tr>
<tr>
<td>compact</td>
<td>boolean</td>
<td>If true, the appender does not use end-of-lines and indentation. Defaults to false.</td>
</tr>
<tr>
<td>eventEol</td>
<td>boolean</td>
<td>If true, the appender appends an end-of-line after each record. Defaults to false. Use with <code>eventEol=true</code> and <code>compact=true</code> to get one record per line.</td>
</tr>
<tr>
<td>complete</td>
<td>boolean</td>
<td>If true, the appender includes the JSON header and footer, and comma between records. Defaults to false.</td>
</tr>
<tr>
<td>properties</td>
<td>boolean</td>
<td>If true, the appender includes the thread context map in the generated JSON. Defaults to false.</td>
</tr>
<tr>
<td>propertiesAsList</td>
<td>boolean</td>
<td>If true, the thread context map is included as a list of map entry objects, where each entry has a &quot;key&quot; attribute (whose value is the key) and a &quot;value&quot; attribute (whose value is the value). Defaults to false, in which case the thread context map is included as a simple map of key-value pairs.</td>
</tr>
<tr>
<td>locationInfo</td>
<td>boolean</td>
<td>If true, the appender includes the location information in the generated JSON. Defaults to false. Generating location information is an expensive operation and may impact performance. Use with caution.</td>
</tr>
<tr>
<td>includeStacktrace</td>
<td>boolean</td>
<td>If true, include full stacktrace of any logged Throwable (optional, default to true).</td>
</tr>
<tr>
<td>stacktraceAsString</td>
<td>boolean</td>
<td>Whether to format the stacktrace as a string, and not a nested object (optional, defaults to false).</td>
</tr>
</tbody>
</table>
10 Layouts

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>includeNullDelimiter</td>
<td>boolean</td>
<td>Whether to include NULL byte as delimiter after each event (optional, default to false).</td>
</tr>
<tr>
<td>objectMessageAsJsonObject</td>
<td>boolean</td>
<td>If true, ObjectMessage is serialized as JSON object to the &quot;message&quot; field of the output log. Defaults to false.</td>
</tr>
</tbody>
</table>

JsonLayout Parameters

To include any custom field in the output, use following syntax:

```
<JsonLayout>
  <KeyValuePair key="additionalField1" value="constant value"/>
  <KeyValuePair key="additionalField2" value="$$\{ctx: key\}"/>
</JsonLayout>
```

Custom fields are always last, in the order they are declared. The values support lookups. Additional runtime dependencies are required for using JsonLayout.

10.1.5 Pattern Layout

A flexible layout configurable with pattern string. The goal of this class is to format a LogEvent and return the results. The format of the result depends on the conversion pattern.

The conversion pattern is closely related to the conversion pattern of the printf function in C. A conversion pattern is composed of literal text and format control expressions called conversion specifiers.

Note that any literal text, including Special Characters, may be included in the conversion pattern. Special Characters include \t, \n, \r, \f. Use \\ to insert a single backslash into the output.

Each conversion specifier starts with a percent sign (%) and is followed by optional format modifiers and a conversion character. The conversion character specifies the type of data, e.g. category, priority, date, thread name. The format modifiers control such things as field width, padding, left and right justification. The following is a simple example.

Let the conversion pattern be "%-5p [%t]: %m%n" and assume that the Log4j environment was set to use a PatternLayout. Then the statements

```
Logger logger = LogManager.getLogger("MyLogger");
logger.debug("Message 1");
logger.warn("Message 2");
```

would yield the output

```
DEBUG [main]: Message 1
WARN  [main]: Message 2
```

Note that there is no explicit separator between text and conversion specifiers. The pattern parser knows when it has reached the end of a conversion specifier when it reads a conversion character. In the example above the conversion specifier %-5p means the priority of the logging event should be left justified to a width of five characters.

If the pattern string does not contain a specifier to handle a Throwable being logged, parsing of the pattern will act as if the "%xEx" specifier had be added to the end of the string. To suppress formatting of the Throwable completely simply add "%ex{0}" as a specifier in the pattern string.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting the syslog String to a byte array. The String must be a valid Charset. If not specified, this layout uses the platform default character set.</td>
</tr>
<tr>
<td>pattern</td>
<td>String</td>
<td>A composite pattern string of one or more conversion patterns from the table below. Cannot be specified with a PatternSelector.</td>
</tr>
<tr>
<td>patternSelector</td>
<td>PatternSelector</td>
<td>A component that analyzes information in the LogEvent and determines which pattern should be used to format the event. The pattern and patternSelector parameters are mutually exclusive.</td>
</tr>
<tr>
<td>replace</td>
<td>RegexReplacement</td>
<td>Allows portions of the resulting String to be replaced. If configured, the replace element must specify the regular expression to match and the substitution. This performs a function similar to the RegexReplacement converter but applies to the whole message while the converter only applies to the String its pattern generates.</td>
</tr>
<tr>
<td>alwaysWriteExceptions</td>
<td>boolean</td>
<td>If true (it is by default) exceptions are always written even if the pattern contains no exception conversions. This means that if you do not include a way to output exceptions in your pattern, the default exception formatter will be added to the end of the pattern. Setting this to false disables this behavior and allows you to exclude exceptions from your pattern output.</td>
</tr>
<tr>
<td>header</td>
<td>String</td>
<td>The optional header string to include at the top of each log file.</td>
</tr>
<tr>
<td>footer</td>
<td>String</td>
<td>The optional footer string to include at the bottom of each log file.</td>
</tr>
<tr>
<td>disableAnsi</td>
<td>boolean</td>
<td>If true (default is false), do not output ANSI escape codes.</td>
</tr>
<tr>
<td>noConsoleNoAnsi</td>
<td>boolean</td>
<td>If true (default is false) and System.console() is null, do not output ANSI escape codes.</td>
</tr>
</tbody>
</table>

PatternLayout Parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex</td>
<td>String</td>
<td>A Java-compliant regular expression to match in the resulting string. See Pattern.</td>
</tr>
<tr>
<td>replacement</td>
<td>String</td>
<td>The string to replace any matched sub-strings with.</td>
</tr>
</tbody>
</table>

**RegexReplacement Parameters**

10.1.5.1 Patterns

The conversions that are provided with Log4j are:

<table>
<thead>
<tr>
<th>Conversion Pattern</th>
<th>Description</th>
</tr>
</thead>
</table>

©2019, The Apache Software Foundation • ALL RIGHTS RESERVED.
### logger\{precision\}

Outputs the name of the logger that published the logging event. The logger conversion specifier can be optionally followed by `precision specifier`, which consists of a decimal integer, or a pattern starting with a decimal integer.

When the precision specifier is an integer value, it reduces the size of the logger name. If the number is positive, the layout prints the corresponding number of rightmost logger name components. If negative, the layout removes the corresponding number of leftmost logger name components.

If the precision contains any non-integer characters, then the layout abbreviates the name based on the pattern. If the precision integer is less than one, the layout still prints the right-most token in full. By default, the layout prints the logger name in full.

<table>
<thead>
<tr>
<th>Conversion Pattern</th>
<th>Logger Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%c{1}</code></td>
<td>org.apache.commons.Foo</td>
<td>Foo</td>
</tr>
<tr>
<td><code>%c{2}</code></td>
<td>org.apache.commons.Foo</td>
<td>commons.Foo</td>
</tr>
<tr>
<td><code>%c{10}</code></td>
<td>org.apache.commons.Foo</td>
<td>org.apache.commons.Foo</td>
</tr>
<tr>
<td><code>%c{-1}</code></td>
<td>org.apache.commons.Foo</td>
<td>apache.commons.Foo</td>
</tr>
<tr>
<td><code>%c{-2}</code></td>
<td>org.apache.commons.Foo</td>
<td>commons.Foo</td>
</tr>
<tr>
<td><code>%c{-10}</code></td>
<td>org.apache.commons.Foo</td>
<td>org.apache.commons.Foo</td>
</tr>
<tr>
<td><code>%c{1.}</code></td>
<td>org.apache.commons.Foo</td>
<td>o.a.c.Foo</td>
</tr>
<tr>
<td><code>%c{1.~.~}</code></td>
<td>org.apache.commons.test.Foo</td>
<td>o.a.<del>.</del>.Foo</td>
</tr>
<tr>
<td><code>%c{.}</code></td>
<td>org.apache.commons.test.Foo</td>
<td>....Foo</td>
</tr>
</tbody>
</table>
| **C(precision)**  
| **class(precision)** | Outputs the fully qualified class name of the caller issuing the logging request. This conversion specifier can be optionally followed by *precision specifier*, that follows the same rules as the logger name converter. 
Generating the class name of the caller (location information) is an expensive operation and may impact performance. Use with caution. |
`d(pattern)`

Outputs the date of the logging event. The date conversion specifier may be followed by a set of braces containing a date and time pattern string per `SimpleDateFormat`.

The predefined named formats are:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%d[DEFAULT]</code></td>
<td>2012-11-02 14:34:02,123</td>
</tr>
<tr>
<td><code>%d[DEFAULT_MICROS]</code></td>
<td>2012-11-02 14:34:02,123456</td>
</tr>
<tr>
<td><code>%d[DEFAULT_NANOS]</code></td>
<td>2012-11-02 14:34:02,123456789</td>
</tr>
<tr>
<td><code>%d[ISO8601]</code></td>
<td>2012-11-02T14:34:02,781</td>
</tr>
<tr>
<td><code>%d[ISO8601_BASIC]</code></td>
<td>20121102T143402,781</td>
</tr>
<tr>
<td><code>%d[ISO8601_OFFSET_DATE_TIME_HH]</code></td>
<td>2012-11-02'T'14:34:02,781-07</td>
</tr>
<tr>
<td><code>%d[ISO8601_OFFSET_DATE_TIME_HHMM]</code></td>
<td>2012-11-02'T'14:34:02,781-0700</td>
</tr>
<tr>
<td><code>%d[ISO8601_OFFSET_DATE_TIME_HHMM]</code></td>
<td>2012-11-02'T'14:34:02,781-07:00</td>
</tr>
<tr>
<td><code>%d[ABSOLUTE]</code></td>
<td>14:34:02,781</td>
</tr>
<tr>
<td><code>%d[ABSOLUTE_MICROS]</code></td>
<td>14:34:02,123456</td>
</tr>
<tr>
<td><code>%d[ABSOLUTE_NANOS]</code></td>
<td>14:34:02,123456789</td>
</tr>
<tr>
<td><code>%d[DATE]</code></td>
<td>02 Nov 2012 14:34:02,781</td>
</tr>
<tr>
<td><code>%d[COMPACT]</code></td>
<td>20121102143402781</td>
</tr>
<tr>
<td><code>%d[UNIX]</code></td>
<td>1351866842</td>
</tr>
<tr>
<td><code>%d[UNIX_MILLIS]</code></td>
<td>1351866842781</td>
</tr>
</tbody>
</table>

You can also use a set of braces containing a time zone id per `java.util.TimeZone.getTimeZone`. If no date format specifier is given then the DEFAULT format is used.

You can define custom date formats:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%d[HH:mm:ss,SSS]</code></td>
<td>14:34:02,123</td>
</tr>
<tr>
<td><code>%d[HH:mm:ss,nnnn] to %d[HH:mm:ss,nnnnnnnn]</code></td>
<td>14:34:02,1234 to 14:34:02,123456789</td>
</tr>
<tr>
<td><code>%d[dd MMM yyyy HH:mm:ss]</code></td>
<td>02 Nov 2012 14:34:02,781</td>
</tr>
<tr>
<td><code>%d[dd MMM yyyy HH:mm:ss]</code></td>
<td>02 Nov 2012 14:34:02,123</td>
</tr>
<tr>
<td><code>%d[dd MMM yyyy HH:mm:ss]</code></td>
<td>02 Nov 2012 14:34:02,1234</td>
</tr>
<tr>
<td><code>%d[dd MMM yyyy HH:mm:ss]</code></td>
<td>02 Nov 2012 14:34:02,123456789</td>
</tr>
<tr>
<td><code>%d[HH:mm:ss][GMT+0]</code></td>
<td>18:34:02</td>
</tr>
</tbody>
</table>

 `%d[UNIX]` outputs the UNIX time in seconds. `%d[UNIX_MILLIS]` outputs the UNIX time in milliseconds. The UNIX time is the difference, in seconds for UNIX and in milliseconds for UNIX_MILLIS, between the current time and midnight, January 1, 1970 UTC. While the time unit is milliseconds, the granularity depends on the operating system (e.g. Windows). This is an efficient way to output the event time because only a conversion from long to String takes place, there is no Date formatting involved.

Log4j 2.11 adds limited support for timestamps more precise than milliseconds when running on Java 9. Note that not all `DateTimeFormatter` formats are supported. Only timestamps in the formats mentioned in the table above may use the "nano-of-second" pattern letter `n` instead of the "fraction-of-second" pattern letter `S`. Users may revert back to a millisecond-precision clock when running on Java 9 by setting system property `log4j2.Clock` to `SystemMillisClock`. 

You should use the predefined named date formats when running on Java 11.
Encodes and escapes special characters suitable for output in specific markup languages. By default, this encodes for HTML if only one option is specified. The second option is used to specify which encoding format should be used. This converter is particularly useful for encoding user provided data so that the output data is not written improperly or insecurely.

A typical usage would encode the message `%enc{%m}` but user input could come from other locations as well, such as the MDC `%enc{%mdc{key}}`

Using the HTML encoding format, the following characters are replaced:

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r', 'n'</td>
<td>Converted into escaped strings &quot;\r&quot; and &quot;\n&quot; respectively</td>
</tr>
<tr>
<td>&amp; , &lt; , &gt; , &quot; , ' /</td>
<td>Replaced with the corresponding HTML entity</td>
</tr>
</tbody>
</table>

Using the XML encoding format, this follows the escaping rules specified by the XML specification:

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; , &lt; , &gt; , &quot; , '</td>
<td>Replaced with the corresponding XML entity</td>
</tr>
</tbody>
</table>

Using the JSON encoding format, this follows the escaping rules specified by RFC 4627 section 2.5:

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0000 - U+001F</td>
<td>\u0000 - \u001F</td>
</tr>
<tr>
<td>Any other control characters</td>
<td>Encoded into its \uABCD equivalent escaped code point</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>\</td>
<td>\ \</td>
</tr>
</tbody>
</table>

For example, the pattern `{"message": "%enc{%m}{JSON}"}` could be used to output a valid JSON document containing the log message as a string value.

Using the CRLF encoding format, the following characters are replaced:

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r', 'n'</td>
<td>Converted into escaped strings &quot;\r&quot; and &quot;\n&quot; respectively</td>
</tr>
</tbody>
</table>
equals(pattern|test|substitution)
equalsIgnoreCase(pattern|test|substitution)

Replaces occurrences of 'test', a string, with its replacement 'substitution' in the string resulting from evaluation of the pattern. For example, "%equals(\[\%marker\]\{\}\)" will replace ']' strings produces by events without markers with an empty string.

The pattern can be arbitrarily complex and in particular can contain multiple conversion keywords.

Outputs the Throwable trace bound to the logging event, by default this will output the full trace as one would normally find with a call to Throwable.printStackTrace().

You can follow the throwable conversion word with an option in the form

%throwable(option).

%throwable(short) outputs the first line of the Throwable.
%throwable(short.className) outputs the name of the class where the exception occurred.
%throwable(short.methodName) outputs the method name where the exception occurred.
%throwable(short.fileName) outputs the name of the class where the exception occurred.
%throwable(short.lineNumber) outputs the line number where the exception occurred.
%throwable(short.message) outputs the message.
%throwable(short.localizedMessage) outputs the localized message.
%throwable(n) outputs the first n lines of the stack trace.

Specifying %throwable(\(none\)) or %throwable(0) suppresses output of the exception.

Use {filters(packages)} where packages is a list of package names to suppress matching stack frames from stack traces.

Use {suffix(pattern)} to add the output of pattern at the end of each stack frames.

Use a {separator(...)} as the end-of-line string. For example: separator(\(\)). The default value is the line.separator system property, which is operating system dependent.
`file` Outputs the file name where the logging request was issued.

Generating the file information (location information) is an expensive operation and may impact performance. Use with caution.

`highlight(pattern)[style]` Adds ANSI colors to the result of the enclosed pattern based on the current event's logging level. (See Jansi configuration.)

The default colors for each level are:

<table>
<thead>
<tr>
<th>Level</th>
<th>ANSI color</th>
</tr>
</thead>
<tbody>
<tr>
<td>FATAL</td>
<td>Bright red</td>
</tr>
<tr>
<td>ERROR</td>
<td>Bright red</td>
</tr>
<tr>
<td>WARN</td>
<td>Yellow</td>
</tr>
<tr>
<td>INFO</td>
<td>Green</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Cyan</td>
</tr>
<tr>
<td>TRACE</td>
<td>Black (looks dark grey)</td>
</tr>
</tbody>
</table>

The color names are ANSI names defined in the `AnsiEscape` class.

The color and attribute names and are standard, but the exact shade, hue, or value.

Color table

You can use the default colors with:

```text
%highlight{%d [%t] %-5level: %msg%n%throwable}
```

You can override the default colors in the optional `{style}` option. For example:

```text
%highlight{%d [%t] %-5level: %msg%n%throwable}

{FATAL=white, ERROR=red, WARN=blue, INFO=black, DEBUG=green, TRACE=blue}
```

You can highlight only a portion of the log event:

```text
%d [%t] %highlight{%-5level: %msg%n%throwable}
```

You can style one part of the message and highlight the rest the log event:

```text
%style{%d [%t]}{black} %highlight{%-5level: %msg%n%throwable}
```
You can also use the `STYLE` key to use a predefined group of colors:

```java
@highlight{@d [tt] %-5level: @msg@n@throwable}
{STYLE=Logback}
```

The `STYLE` value can be one of:

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>See above</td>
</tr>
<tr>
<td>Logback</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level</th>
<th>ANSI color</th>
</tr>
</thead>
<tbody>
<tr>
<td>FATAL</td>
<td>Blinking bright red</td>
</tr>
<tr>
<td>ERROR</td>
<td>Bright red</td>
</tr>
<tr>
<td>WARN</td>
<td>Red</td>
</tr>
<tr>
<td>INFO</td>
<td>Blue</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Normal</td>
</tr>
<tr>
<td>TRACE</td>
<td>Normal</td>
</tr>
</tbody>
</table>

K{key}

`map{key}`

`MAP{key}`

Outputs the entries in a `MapMessage`, if one is present in the event. The K conversion character can be followed by the key for the map placed between braces, as in `%K{clientNumber}` where `clientNumber` is the key. The value in the Map corresponding to the key will be output. If no additional sub-option is specified, then the entire contents of the Map key value pair set is output using a format `{{key1,val1},{key2,val2}}`.

I

`location`

Outputs location information of the caller which generated the logging event.

The location information depends on the JVM implementation but usually consists of the fully qualified name of the calling method followed by the callers source file name and line number between parentheses.

Generating `location information` is an expensive operation and may impact performance. Use with caution.

L

`line`

Outputs the line number from where the logging request was issued.

Generating line number information (`location information`) is an expensive operation and may impact performance. Use with caution.
msg[nolookups]{ansi} Outputs the application supplied message associated with the logging event.

Add {ansi} to render messages with ANSI escape codes (requires JAnsi, see configuration.)

The default syntax for embedded ANSI codes is:

```
@| code(, code) * text |@
```

For example, to render the message "Hello" in green, use:

```
@|green Hello|@
```

To render the message "Hello" in bold and red, use:

```
@|bold,red Warning!|@
```

You can also define custom style names in the configuration with the syntax:

```
%message{ansi}{StyleName=value(,value)*( StyleName=value(,value)*)}n
```

For example:

```
%message{ansi}{WarningStyle=red,bold KeyStyle=white ValueStyle=blue}n
```

The call site can look like this:

```
logger.info("@|KeyStyle {}|@ = @|ValueStyle {}|@", entry.getKey(), entry.getValue());
```

Use {nolookups} to log messages like "${esc.d}{date:YYYY-MM-dd}" without using any lookups. Normally calling `logger.info("Try ${esc.d}{date:YYYY-MM-dd}"'); would replace the date template with an actual date. Using nolookups disables this feature and logs the message string untouched.

method

Outputs the method name where the logging request was issued.

Generating the method name of the caller (location information) is an expensive operation and may impact performance. Use with caution.

marker

The full name of the marker, including parents, if one is present.

markerSimpleName

The simple name of the marker (not including parents), if one is present.
| **maxLen** | Outputs the result of evaluating the pattern and truncating the result. If the length is greater than 20, then the output will contain a trailing ellipsis. If the provided length is invalid, a default value of 100 is used. |
| **maxLength** |
| Example syntax: `%maxLen{%p: %c{1} - %m %notEmpty{ =>$%ex{short}}}{160}` will be limited to 160 characters with a trailing ellipsis. Another example: `%maxLen{%m}{20}` will be limited to 20 characters and no trailing ellipsis. |

| **n** | Outputs the platform dependent line separator character or characters. |
| This conversion character offers practically the same performance as using non-portable line separator strings such as "\n", or "\r\n". Thus, it is the preferred way of specifying a line separator. |

| **N** | Outputs the result of `System.nanoTime()` at the time the log event was created. |
| **nano** |

| **pid**[defaultValue] | Outputs the process ID if supported by the underlying platform. An optional default value may be specified to be shown if the platform does not support process IDs. |
| **processId**[defaultValue] |

| **variablesNotEmpty(pattern)** | Outputs the result of evaluating the pattern if and only if all variables in the pattern are not empty. |
| **varsNotEmpty(pattern)** |
| **notEmpty(pattern)** |
| For example: |
| `%notEmpty{[%marker]}` |
| **p** | **level** | Outputs the level of the logging event.  
You provide a level name map in the form "level=value, level=value" where level is the name of the Level and value is the value that should be displayed instead of the name of the Level.

For example:
```plaintext
%level{WARN=Warning, DEBUG=Debug, ERROR=Error, TRACE=Trace, INFO=Info}
```
Alternatively, for the compact-minded:
```plaintext
%level{WARN=W, DEBUG=D, ERROR=E, TRACE=T, INFO=I}
```
More succinctly, for the same result as above, you can define the length of the level label:
```plaintext
%level{length=1}
```
If the length is greater than a level name length, the layout uses the normal level name.

You can combine the two kinds of options:
```plaintext
%level{ERROR=Error, length=2}
```
This give you the Error level name and all other level names of length 2.

Finally, you can output lower-case level names (the default is upper-case):
```plaintext
%level{lowerCase=true}
```
| **r** | **relative** | Outputs the number of milliseconds elapsed since the JVM was started until the creation of the logging event.

| **replace** | **pattern** | **regex** | **substitution** | Replaces occurrences of 'regex', a regular expression, with its replacement 'substitution' in the string resulting from evaluation of the pattern. For example, "%replace{\%msg}{\s}\{\}\" will remove all spaces contained in the event message.

The pattern can be arbitrarily complex and in particular can contain multiple conversion keywords. For instance, "%replace{\%logger\%msg}{\\}\{/\}\{" will replace all dots in the logger or the message of the event with a forward slash.
The same as the \%throwable conversion word but the stack trace is printed starting with the first exception that was thrown followed by each subsequent wrapping exception.

The throwable conversion word can be followed by an option in the form \%rEx(short) which will only output the first line of the Throwable or \%rEx(n) where the first n lines of the stack trace will be printed.

Specifying \%rEx(none) or \%rEx(0) will suppress printing of the exception.

Use filters(packages) where packages is a list of package names to suppress matching stack frames from stack traces.

Use a separator string to separate the lines of a stack trace. For example: separator(|). The default value is the line.separator system property, which is operating system dependent.

Use rEx(suffix(pattern)) to add the output of pattern to the output only when there is a throwable to print.

Includes a sequence number that will be incremented in every event. The counter is a static variable so will only be unique within applications that share the same converter Class object.
**style(pattern)(ANSI style)**

Uses ANSI escape sequences to style the result of the enclosed pattern. The style can consist of a comma separated list of style names from the following table. (See Jansi configuration.)

<table>
<thead>
<tr>
<th>Style Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Normal display</td>
</tr>
<tr>
<td>Bright</td>
<td>Bold</td>
</tr>
<tr>
<td>Dim</td>
<td>Dimmed or faint characters</td>
</tr>
<tr>
<td>Underline</td>
<td>Underlined characters</td>
</tr>
<tr>
<td>Blink</td>
<td>Blinking characters</td>
</tr>
<tr>
<td>Reverse</td>
<td>Reverse video</td>
</tr>
<tr>
<td>Hidden</td>
<td>Black or FG_Black</td>
</tr>
<tr>
<td>BG_Black</td>
<td>Set background color to black</td>
</tr>
<tr>
<td>Red or FG_Red</td>
<td>Set foreground color to red</td>
</tr>
<tr>
<td>Green or FG_Green</td>
<td>Set foreground color to green</td>
</tr>
<tr>
<td>Yellow or FG_Yellow</td>
<td>Set foreground color to yellow</td>
</tr>
<tr>
<td>Blue or FG_Blue</td>
<td>Set foreground color to blue</td>
</tr>
<tr>
<td>Magenta or FG_Magenta</td>
<td>Set foreground color to magenta</td>
</tr>
<tr>
<td>Cyan or FG_Cyan</td>
<td>Set foreground color to cyan</td>
</tr>
<tr>
<td>White or FG_White</td>
<td>Set foreground color to white</td>
</tr>
<tr>
<td>BG_Black</td>
<td>Set background color to black</td>
</tr>
<tr>
<td>BG_Red</td>
<td>Set background color to red</td>
</tr>
<tr>
<td>BG_Green</td>
<td>Set background color to green</td>
</tr>
<tr>
<td>BG_Yellow</td>
<td>Set background color to yellow</td>
</tr>
<tr>
<td>BG_Blue</td>
<td>Set background color to blue</td>
</tr>
<tr>
<td>BG_Magenta</td>
<td>Set background color to magenta</td>
</tr>
<tr>
<td>BG_Cyan</td>
<td>Set background color to cyan</td>
</tr>
<tr>
<td>BG_White</td>
<td>Set background color to white</td>
</tr>
</tbody>
</table>

For example:

```
%d{ISO8601} %style{black} %t{blue} %-5level: %msg%n %throwable
```

You can also combine styles:

```
%d %highlight{%p} %style{%logger}{bright,cyan} %C{1.} %msg%n
```

You can also use % with color like %black.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T</code></td>
<td>Outputs the ID of the thread that generated the logging event.</td>
</tr>
<tr>
<td><code>tid</code></td>
<td>Outputs the name of the thread that generated the logging event.</td>
</tr>
<tr>
<td><code>t</code></td>
<td>Outputs the priority of the thread that generated the logging event.</td>
</tr>
<tr>
<td><code>tn</code></td>
<td>Outputs the fully qualified class name of the logger.</td>
</tr>
<tr>
<td><code>threadName</code></td>
<td>Outputs the EndOfBatch status of the logging event, as &quot;true&quot; or &quot;false&quot;.</td>
</tr>
<tr>
<td><code>tp</code></td>
<td>Outputs the Thread Context Stack (also known as the Nested Diagnostic Context or NDC) associated with the thread that generated the logging event.</td>
</tr>
<tr>
<td><code>fqcn</code></td>
<td>Outputs the Thread Context Map (also known as the Mapped Diagnostic Context or MDC) associated with the thread that generated the logging event. The X conversion character can be followed by one or more keys for the map placed between braces, as in <code>%X{clientNumber}</code> where <code>clientNumber</code> is the key. The value in the MDC corresponding to the key will be output.</td>
</tr>
<tr>
<td><code>x</code></td>
<td>If a list of keys are provided, such as <code>%X{name, number}</code>, then each key that is present in the ThreadContext will be output using the format <code>{name=val1, number=val2}</code>. The key/value pairs will be printed in the order they appear in the list.</td>
</tr>
<tr>
<td><code>mdc</code></td>
<td>If no sub-options are specified then the entire contents of the MDC key value pair set is output using a format <code>{key1=val1, key2=val2}</code>. The key/value pairs will be printed in sorted order.</td>
</tr>
<tr>
<td><code>MDC</code></td>
<td>See the ThreadContext class for more details.</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Includes either a random or a time-based UUID. The time-based UUID is a Type 1 UUID that can generate up to 10,000 unique ids per millisecond, will use the MAC address of each host, and to try to insure uniqueness across multiple JVMs and/or ClassLoaders on the same host a random number between 0 and 16,384 will be associated with each instance of the UUID generator Class and included in each time-based UUID generated. Because time-based UUIDs contain the MAC address and timestamp they should be used with care as they can cause a security vulnerability.</td>
</tr>
</tbody>
</table>

**Note:** Time-based UUIDs contain the MAC address and timestamp and should be used with care as they can cause a security vulnerability.
The same as the %throwable conversion word but also includes class packaging information.

At the end of each stack element of the exception, a string containing the name of the jar file that contains the class or the directory the class is located in and the "Implementation-Version" as found in that jar’s manifest will be added. If the information is uncertain, then the class packaging data will be preceded by a tilde, i.e. the ‘~’ character.

The throwable conversion word can be followed by an option in the form %xEx{short} which will only output the first line of the Throwable or %xEx{n} where the first n lines of the stack trace will be printed. Specifying %xEx{none} or %xEx{0} will suppress printing of the exception.

Use filters(packages) where packages is a list of package names to suppress matching stack frames from stack traces.

Use a separator string to separate the lines of a stack trace. For example: separator(|). The default value is the line.separator system property, which is operating system dependent.

The ansi option renders stack traces with ANSI escapes code using the JAnsi library. (See configuration.) Use {ansi} to use the default color mapping. You can specify your own mappings with key=value pairs. The keys are:

- Prefix
- Name
- NameMessageSeparator
- Message
- At
- CauseLabel
- Text
- More
- Suppressed
- StackTraceElement.ClassName
- StackTraceElement.ClassMethodSeparator
- StackTraceElement.MethodName
- StackTraceElement.NativeMethod
- StackTraceElement.FileName
- StackTraceElementLineNumber
- StackTraceElement.Container
- StackTraceElement.ContainerSeparator
- StackTraceElement.UnknownSource
- ExtraClassInfo.Inexact
- ExtraClassInfo.Container
- ExtraClassInfo.ContainerSeparator
- ExtraClassInfo.Location
- ExtraClassInfo.Version

The values are names from JAnsi’s Code class like blue, bg_red, and so on (Log4j ignores case.)

The special key StyleMapName can be set to
The sequence `%%` outputs a single percent sign.

By default the relevant information is output as is. However, with the aid of format modifiers it is possible to change the minimum field width, the maximum field width and justification.

The optional format modifier is placed between the percent sign and the conversion character.

The first optional format modifier is the **left justification flag** which is just the minus (-) character. Then comes the optional **minimum field width** modifier. This is a decimal constant that represents the minimum number of characters to output. If the data item requires fewer characters, it is padded on either the left or the right until the minimum width is reached. The default is to pad on the left (right justify) but you can specify right padding with the left justification flag. The padding character is space. If the data item is larger than the minimum field width, the field is expanded to accommodate the data. The value is never truncated.

This behavior can be changed using the **maximum field width** modifier which is designated by a period followed by a decimal constant. If the data item is longer than the maximum field, then the extra characters are removed from the **beginning** of the data item and not from the end. For example, if the maximum field width is eight and the data item is ten characters long, then the first two characters of the data item are dropped. This behavior deviates from the printf function in C where truncation is done from the end.

Truncation from the end is possible by appending a minus character right after the period. In that case, if the maximum field width is eight and the data item is ten characters long, then the last two characters of the data item are dropped.

Below are various format modifier examples for the category conversion specifier.

<table>
<thead>
<tr>
<th>Format modifier</th>
<th>left justify</th>
<th>minimum width</th>
<th>maximum width</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%20c</td>
<td>false</td>
<td>20</td>
<td>none</td>
<td>Left pad with spaces if the category name is less than 20 characters long.</td>
</tr>
<tr>
<td>%-20c</td>
<td>true</td>
<td>20</td>
<td>none</td>
<td>Right pad with spaces if the category name is less than 20 characters long.</td>
</tr>
<tr>
<td>%.30c</td>
<td>NA</td>
<td>none</td>
<td>30</td>
<td>Truncate from the beginning if the category name is longer than 30 characters.</td>
</tr>
<tr>
<td>%20.30c</td>
<td>false</td>
<td>20</td>
<td>30</td>
<td>Left pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the beginning.</td>
</tr>
</tbody>
</table>
Pattern Converters

10.1.5.2 ANSI Styling on Windows
ANSI escape sequences are supported natively on many platforms but are not by default on Windows. To enable ANSI support add the Jansi jar to your application and set property log4j.skipJansi to false. This allows Log4j to use Jansi to add ANSI escape codes when writing to the console.

NOTE: Prior to Log4j 2.10, Jansi was enabled by default. The fact that Jansi requires native code means that Jansi can only be loaded by a single class loader. For web applications this means the Jansi jar has to be in the web container's classpath. To avoid causing problems for web applications, Log4j will no longer automatically try to load Jansi without explicit configuration from Log4j 2.10 onward.

10.1.5.3 Example Patterns

10.Filtered Throwables
This example shows how to filter out classes from unimportant packages in stack traces.

<properties>
  <property name="filters">org.junit,org.apache.maven,sun.reflect,java.lang.reflect</property>
</properties>
...

The result printed to the console will appear similar to:
Exception java.lang.IllegalArgumentException: IllegalArgumentException
   at org.apache.logging.log4j.core.pattern.ExtendedThrowableTest.
   testException(ExtendedThrowableTest.java:72) [test-classes/:?]
   ... suppressed 26 lines
   at $Proxy0.invoke(Unknown Source)} [?:?]
   ... suppressed 3 lines
   Caused by: java.lang.NullPointerException: null pointer
   at org.apache.logging.log4j.core.pattern.ExtendedThrowableTest.
   testException(ExtendedThrowableTest.java:71) ~[test-classes/:?]
   ... 30 more

10. ANSI Styled
The log level will be highlighted according to the event's log level. All the content that follows the
level will be bright green.

<PatternLayout>
   <pattern>%d %highlight{p} %style{%C{1.} [%t] %m}{bold,green}%n</pattern>
</PatternLayout>

10.1.5.4 Pattern Selectors
The PatternLayout can be configured with a PatternSelector to allow it to choose a pattern to
use based on attributes of the log event or other factors. A PatternSelector will normally be
configured with a defaultPattern attribute, which is used when other criteria don't match, and a set of
PatternMatch elements that identify the various patterns that can be selected.

10. MarkerPatternSelector
The MarkerPatternSelector selects patterns based on the Marker included in the log event. If the
Marker in the log event is equal to or is an ancestor of the name specified on the PatternMatch key
attribute, then the pattern specified on that PatternMatch element will be used.

<PatternLayout>
   <MarkerPatternSelector defaultPattern="[%-5level] %c{1.} %msg%n">
      <PatternMatch key="FLOW" pattern="[%-5level] %c{1.} ====== %C{1.}.%M:%L %msg ======%n"/>
   </MarkerPatternSelector>
</PatternLayout>

10. ScriptPatternSelector
The ScriptPatternSelector executes a script as described in the Scripts section of the Configuration
chapter. The script is passed all the properties configured in the Properties section of the
configuration, the StrSubstitutor used by the Configuration in the "substitutor" variables, and the log
event in the "logEvent" variable, and is expected to return the value of the PatternMatch key that
should be used, or null if the default pattern should be used.
10.1.6 RFC5424 Layout

As the name implies, the Rfc5424Layout formats LogEvents in accordance with RFC 5424, the enhanced Syslog specification. Although the specification is primarily directed at sending messages via Syslog, this format is quite useful for other purposes since items are passed in the message as self-describing key/value pairs.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appName</td>
<td>String</td>
<td>The value to use as the APP-NAME in the RFC 5424 syslog record.</td>
</tr>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting the syslog String to a byte array. The String must be a valid Charset. If not specified, the default system Charset will be used.</td>
</tr>
<tr>
<td>enterpriseNumber</td>
<td>integer</td>
<td>The IANA enterprise number as described in RFC 5424</td>
</tr>
<tr>
<td>exceptionPattern</td>
<td>String</td>
<td>One of the conversion specifiers from PatternLayout that defines which ThrowablePatternConverter to use to format exceptions. Any of the options that are valid for those specifiers may be included. The default is to not include the Throwable from the event, if any, in the output.</td>
</tr>
<tr>
<td><strong>facilit</strong></td>
<td><strong>String</strong></td>
<td>The facility is used to try to classify the message. The facility option must be set to one of &quot;KERN&quot;, &quot;USER&quot;, &quot;MAIL&quot;, &quot;DAEMON&quot;, &quot;AUTH&quot;, &quot;SYSLOG&quot;, &quot;LPR&quot;, &quot;NEWS&quot;, &quot;UUCP&quot;, &quot;CRON&quot;, &quot;AUTHPRIV&quot;, &quot;FTP&quot;, &quot;NTP&quot;, &quot;AUDIT&quot;, &quot;ALERT&quot;, &quot;CLOCK&quot;, &quot;LOCAL0&quot;, &quot;LOCAL1&quot;, &quot;LOCAL2&quot;, &quot;LOCAL3&quot;, &quot;LOCAL4&quot;, &quot;LOCAL5&quot;, &quot;LOCAL6&quot;, or &quot;LOCAL7&quot;. These values may be specified as upper or lower case characters.</td>
</tr>
<tr>
<td><strong>format</strong></td>
<td><strong>String</strong></td>
<td>If set to &quot;RFC5424&quot; the data will be formatted in accordance with RFC 5424. Otherwise, it will be formatted as a BSD Syslog record. Note that although BSD Syslog records are required to be 1024 bytes or shorter the SyslogLayout does not truncate them. The RFC5424Layout also does not truncate records since the receiver must accept records of up to 2048 bytes and may accept records that are longer.</td>
</tr>
<tr>
<td><strong>id</strong></td>
<td><strong>String</strong></td>
<td>The default structured data id to use when formatting according to RFC 5424. If the LogEvent contains a StructuredDataMessage the id from the Message will be used instead of this value.</td>
</tr>
<tr>
<td><strong>includeMDC</strong></td>
<td><strong>boolean</strong></td>
<td>Indicates whether data from the ThreadContextMap will be included in the RFC 5424 Syslog record. Defaults to true.</td>
</tr>
<tr>
<td><strong>loggerFields</strong></td>
<td><strong>List of KeyValuePairs</strong></td>
<td>Allows arbitrary PatternLayout patterns to be included as specified ThreadContext fields; no default specified. To use, include a &lt;LoggerFields&gt; nested element, containing one or more &lt;KeyValuePair&gt; elements. Each &lt;KeyValuePair&gt; must have a key attribute, which specifies the key name which will be used to identify the field within the MDC Structured Data element, and a value attribute, which specifies the PatternLayout pattern to use as the value.</td>
</tr>
<tr>
<td><strong>mdcExcludes</strong></td>
<td><strong>String</strong></td>
<td>A comma separated list of mdc keys that should be excluded from the LogEvent. This is mutually exclusive with the mdcIncludes attribute. This attribute only applies to RFC 5424 syslog records.</td>
</tr>
</tbody>
</table>
### Rfc5424Layout Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdcIncludes</td>
<td>String</td>
<td>A comma separated list of mdc keys that should be included in the FlumeEvent. Any keys in the MDC not found in the list will be excluded. This option is mutually exclusive with the mdcExcludes attribute. This attribute only applies to RFC 5424 syslog records.</td>
</tr>
<tr>
<td>mdcRequired</td>
<td>String</td>
<td>A comma separated list of mdc keys that must be present in the MDC. If a key is not present a LoggingException will be thrown. This attribute only applies to RFC 5424 syslog records.</td>
</tr>
<tr>
<td>mdcPrefix</td>
<td>String</td>
<td>A string that should be prepended to each MDC key in order to distinguish it from event attributes. The default string is &quot;mdc:&quot;. This attribute only applies to RFC 5424 syslog records.</td>
</tr>
<tr>
<td>mdcId</td>
<td>String</td>
<td>A required MDC ID. This attribute only applies to RFC 5424 syslog records.</td>
</tr>
<tr>
<td>messageId</td>
<td>String</td>
<td>The default value to be used in the MSGID field of RFC 5424 syslog records.</td>
</tr>
<tr>
<td>newLine</td>
<td>boolean</td>
<td>If true, a newline will be appended to the end of the syslog record. The default is false.</td>
</tr>
<tr>
<td>newLineEscape</td>
<td>String</td>
<td>String that should be used to replace newlines within the message text.</td>
</tr>
</tbody>
</table>

#### 10.1.7 Serialized Layout

The SerializedLayout simply serializes the LogEvent into a byte array using Java Serialization. The SerializedLayout accepts no parameters.

This layout is deprecated since version 2.9. Java Serialization has inherent security weaknesses, using this layout is no longer recommended. An alternative layout containing the same information is JsonLayout, configured with properties="true".

#### 10.1.8 Syslog Layout

The SyslogLayout formats the LogEvent as BSD Syslog records matching the same format used by Log4j 1.2.
## SyslogLayout Parameters

### 10.1.9 XML Layout

Appends a series of Event elements as defined in the [log4j.dtd](http://logging.apache.org/log4j/2.0/events).

**10.1.9.1 Complete well-formed XML vs. fragment XML**

If you configure `complete="true"`, the appender outputs a well-formed XML document where the default namespace is the Log4j namespace "http://logging.apache.org/log4j/2.0/events". By default, with `complete="false"`, you should include the output as an external entity in a separate file to form a well-formed XML document, in which case the appender uses `namespacePrefix` with a default of "log4j".

A well-formed XML document follows this pattern:
<Event xmlns="http://logging.apache.org/log4j/2.0/events"
  level="INFO"
  loggerName="HelloWorld"
  endOfBatch="false"
  thread="main"
  loggerFqcn="org.apache.logging.log4j.spi.AbstractLogger"
  threadId="1"
  threadPriority="5">
  <Instant epochSecond="1493121664" nanoOfSecond="118000000"/>
  <Marker name="child">
    <Parents>
      <Marker name="parent">
        <Parents>
          <Marker name="grandparent"/>
        </Parents>
      </Marker>
    </Parents>
  </Marker>
  <Message>Hello, world!</Message>
  <ContextMap>
    <item key="bar" value="BAR"/>
    <item key="foo" value="FOO"/>
  </ContextMap>
  <ContextStack>
    <ContextStackItem>one</ContextStackItem>
    <ContextStackItem>two</ContextStackItem>
  </ContextStack>
  <Source
    class="logtest.Main"
    method="main"
    file="Main.java"
    line="29"/>
  <Thrown commonElementCount="0" message="error message" name="java.lang.RuntimeException">
    <ExtendedStackTrace>
      <ExtendedStackTraceItem
        class="logtest.Main"
        method="main"
        file="Main.java"
        line="29"
        exact="true"
        location="classes/"
        version="?"/>
    </ExtendedStackTrace>
  </Thrown>
</Event>

If `complete="false"`, the appender does not write the XML processing instruction and the root element.

10.1.9.2 Marker

Markers are represented by a `Marker` element within the `Event` element. The `Marker` element appears only when a marker is used in the log message. The name of the marker's parent will be provided in the `parent` attribute of the `Marker` element.
10.1.9.3 Pretty vs. compact XML

By default, the XML layout is not compact (a.k.a. not "pretty") with \texttt{compact="false"}, which means the appender uses end-of-line characters and indents lines to format the XML. If \texttt{compact="true"}, then no end-of-line or indentation is used. Message content may contain, of course, end-of-lines.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting to a byte array. The value must be a valid \texttt{Charset}. If not specified, UTF-8 will be used.</td>
</tr>
<tr>
<td>compact</td>
<td>boolean</td>
<td>If true, the appender does not use end-of-lines and indentation. Defaults to false.</td>
</tr>
<tr>
<td>complete</td>
<td>boolean</td>
<td>If true, the appender includes the XML header and footer. Defaults to false.</td>
</tr>
<tr>
<td>properties</td>
<td>boolean</td>
<td>If true, the appender includes the thread context map in the generated XML. Defaults to false.</td>
</tr>
<tr>
<td>locationInfo</td>
<td>boolean</td>
<td>If true, the appender includes the location information in the generated XML. Defaults to false. Generating location information is an expensive operation and may impact performance. Use with caution.</td>
</tr>
<tr>
<td>includeStacktrace</td>
<td>boolean</td>
<td>If true, include full stacktrace of any logged \texttt{Throwable} (optional, default to true).</td>
</tr>
<tr>
<td>stacktraceAsString</td>
<td>boolean</td>
<td>Whether to format the stacktrace as a string, and not a nested object (optional, defaults to false).</td>
</tr>
<tr>
<td>includeNullDelimiter</td>
<td>boolean</td>
<td>Whether to include NULL byte as delimiter after each event (optional, default to false).</td>
</tr>
</tbody>
</table>

XmlLayout Parameters

To include any custom field in the output, use following syntax:

```xml
<XmlLayout>
  <KeyValuePair key="additionalField1" value="constant value"/>
  <KeyValuePair key="additionalField2" value="$\{ctx:key\}"/>
</XmlLayout>
```

Custom fields are always last, in the order they are declared. The values support lookups. Additional runtime dependencies are required for using XmlLayout.
### 10.1.10 YAML Layout

Appends a series of YAML events as strings serialized as bytes.

A YAML log event follows this pattern:

```yaml
---
instant:
  epochSecond: 1493121664
  nanoOfSecond: 118000000
thread: "main"
level: "INFO"
loggerName: "HelloWorld"
marker:
  name: "child"
  parents:
    - name: "parent"
      parents:
        - name: "grandparent"
message: "Hello, world!"
thrown:
  commonElementCount: 0
  message: "error message"
  name: "java.lang.RuntimeException"
extendedStackTrace:
- class: "logtest.Main"
  method: "main"
  file: "Main.java"
  line: 29
  exact: true
  location: "classes/"
  version: "?"
contextStack:
- "one"
- "two"
endOfBatch: false
loggerFqcn: "org.apache.logging.log4j.spi.AbstractLogger"
contextMap:
  bar: "BAR"
  foo: "FOO"
threadId: 1
threadPriority: 5
source:
  class: "logtest.Main"
  method: "main"
  file: "Main.java"
  line: 29
```

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>thread</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>level</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>loggerName</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>marker</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>message</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>thrown</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>extendedStack</td>
<td>list</td>
<td></td>
</tr>
<tr>
<td>contextStack</td>
<td>list</td>
<td></td>
</tr>
<tr>
<td>loggerFqcn</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>contextMap</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>threadId</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>threadPriority</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>charset</td>
<td>String</td>
<td>The character set to use when converting to a byte array. The value must be a valid <code>Charset</code>. If not specified, UTF-8 will be used.</td>
</tr>
<tr>
<td>properties</td>
<td>boolean</td>
<td>If true, the appender includes the thread context map in the generated YAML. Defaults to false.</td>
</tr>
<tr>
<td>locationInfo</td>
<td>boolean</td>
<td>If true, the appender includes the location information in the generated YAML. Defaults to false.</td>
</tr>
<tr>
<td>includeStacktrace</td>
<td>boolean</td>
<td>If true, include full stacktrace of any logged <code>Throwable</code> (optional, default to true).</td>
</tr>
<tr>
<td>stacktraceAsString</td>
<td>boolean</td>
<td>Whether to format the stacktrace as a string, and not a nested object (optional, defaults to false).</td>
</tr>
<tr>
<td>includeNullDelimiter</td>
<td>boolean</td>
<td>Whether to include NULL byte as delimiter after each event (optional, default to false).</td>
</tr>
</tbody>
</table>

YamlLayout Parameters

To include any custom field in the output, use following syntax:

```
<YamlLayout>
  <KeyValuePair key="additionalField1" value="constant value"/>
  <KeyValuePair key="additionalField2" value="${ctx:key}"/>
</YamlLayout>
```

Custom fields are always last, in the order they are declared. The values support lookups. Additional runtime dependencies are required for using YamlLayout.

10.1.11 Location Information

If one of the layouts is configured with a location-related attribute like HTML `locationInfo`, or one of the patterns `%C` or `%class`, `%F` or `%file`, `%l` or `%location`, `%L` or `%line`, `%M` or `%method`, Log4j will take a snapshot of the stack, and walk the stack trace to find the location information.

This is an expensive operation: 1.3 - 5 times slower for synchronous loggers. Synchronous loggers wait as long as possible before they take this stack snapshot. If no location is required, the snapshot will never be taken.

However, asynchronous loggers need to make this decision before passing the log message to another thread; the location information will be lost after that point. The performance impact of taking a stack trace snapshot is even higher for asynchronous loggers: logging with location is 30-100 times slower.
than without location. For this reason, asynchronous loggers and asynchronous appenders do not include location information by default.

You can override the default behaviour in your logger or asynchronous appender configuration by specifying `includeLocation="true"`. 
11 Filters

11.1 Filters

Filters allow Log Events to be evaluated to determine if or how they should be published. A Filter will be called on one of its filter methods and will return a Result, which is an Enum that has one of 3 values - ACCEPT, DENY or NEUTRAL.

Filters may be configured in one of four locations:

1. Context-wide Filters are configured directly in the configuration. Events that are rejected by these filters will not be passed to loggers for further processing. Once an event has been accepted by a Context-wide filter it will not be evaluated by any other Context-wide Filters nor will the Logger’s Level be used to filter the event. The event will be evaluated by Logger and Appender Filters however.

2. Logger Filters are configured on a specified Logger. These are evaluated after the Context-wide Filters and the Log Level for the Logger. Events that are rejected by these filters will be discarded and the event will not be passed to a parent Logger regardless of the additivity setting.

3. Appender Filters are used to determine if a specific Appender should handle the formatting and publication of the event.

4. Appender Reference Filters are used to determine if a Logger should route the event to an appender.

11.1.1 BurstFilter

The BurstFilter provides a mechanism to control the rate at which LogEvents are processed by silently discarding events after the maximum limit has been reached.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>String</td>
<td>Level of messages to be filtered. Anything at or below this level will be filtered out if maxBurst has been exceeded. The default is WARN meaning any messages that are higher than warn will be logged regardless of the size of a burst.</td>
</tr>
<tr>
<td>rate</td>
<td>float</td>
<td>The average number of events per second to allow.</td>
</tr>
<tr>
<td>maxBurst</td>
<td>integer</td>
<td>The maximum number of events that can occur before events are filtered for exceeding the average rate. The default is 10 times the rate.</td>
</tr>
<tr>
<td>onMatch</td>
<td>String</td>
<td>Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.</td>
</tr>
</tbody>
</table>
onMismatch | String | Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.

Burst Filter Parameters

A configuration containing the BurstFilter might look like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <BurstFilter level="INFO" rate="16" maxBurst="100"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

11.1.2 CompositeFilter

The CompositeFilter provides a way to specify more than one filter. It is added to the configuration as a filters element and contains other filters to be evaluated. The filters element accepts no parameters.

A configuration containing the CompositeFilter might look like:
<xml version="1.0" encoding="UTF-8"/>
<Configuration status="warn" name="MyApp" packages="">
    <Filters>
        <MarkerFilter marker="EVENT" onMatch="ACCEPT" onMismatch="NEUTRAL"/>
        <DynamicThresholdFilter key="loginId" defaultThreshold="ERROR"
            onMatch="ACCEPT" onMismatch="NEUTRAL">
            <KeyValuePair key="User1" value="DEBUG"/>
        </DynamicThresholdFilter>
    </Filters>
    <Appenders>
        <File name="Audit" fileName="logs/audit.log">
            <PatternLayout>
                <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
        </File>
        <RollingFile name="RollingFile" fileName="logs/app.log"
            filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
            <BurstFilter level="INFO" rate="16" maxBurst="100"/>
            <PatternLayout>
                <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <TimeBasedTriggeringPolicy />
        </RollingFile>
    </Appenders>
    <Loggers>
        <Logger name="EventLogger" level="info">
            <AppenderRef ref="Audit"/>
        </Logger>
        <Root level="error">
            <AppenderRef ref="RollingFile"/>
        </Root>
    </Loggers>
</Configuration>

### 11.1.3 DynamicThresholdFilter

The DynamicThresholdFilter allows filtering by log level based on specific attributes. For example, if the user's loginId is being captured in the ThreadContext Map then it is possible to enable debug logging for only that user. If the log event does not contain the specified ThreadContext item NEUTRAL will be returned.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String</td>
<td>The name of the item in the ThreadContext Map to compare.</td>
</tr>
<tr>
<td>defaultThreshold</td>
<td>String</td>
<td>Level of messages to be filtered. The default threshold only applies if the log event contains the specified ThreadContext Map item and its value does not match any key in the key/value pairs.</td>
</tr>
</tbody>
</table>
Dynamic Threshold Filter Parameters

Here is a sample configuration containing the DynamicThresholdFilter:

```xml
<Configuration status="warn" name="MyApp" packages="">
  <DynamicThresholdFilter key="loginId" defaultThreshold="ERROR"
    onMatch="ACCEPT" onMismatch="NEUTRAL">
    <KeyValuePair key="User1" value="DEBUG"/>
  </DynamicThresholdFilter>
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log"
      filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <BurstFilter level="INFO" rate="16" maxBurst="100"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

11.1.4 MapFilter

The MapFilter allows filtering against data elements that are in a MapMessage.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyValuePair</td>
<td>KeyValuePair[]</td>
<td>One or more KeyValuePair elements that define the key in the map and the value to match on. If the same key is specified more than once then the check for that key will automatically be an &quot;or&quot; since a Map can only contain a single value.</td>
</tr>
</tbody>
</table>
### Map Filter Parameters

As in this configuration, the MapFilter can be used to log particular events:

```xml
<Configuration status="warn" name="MyApp" packages="">
  <MapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
    <KeyValuePair key="eventId" value="Login"/>
    <KeyValuePair key="eventId" value="Logout"/>
  </MapFilter>
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <BurstFilter level="INFO" rate="16" maxBurst="100"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

This sample configuration will exhibit the same behavior as the preceding example since the only logger configured is the root.
This third sample configuration will exhibit the same behavior as the preceding examples since the only logger configured is the root and the root is only configured with a single appender reference.
11.1.5 MarkerFilter

The MarkerFilter compares the configured Marker value against the Marker that is included in the LogEvent. A match occurs when the Marker name matches either the Log Event's Marker or one of its parents.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>marker</td>
<td>String</td>
<td>The name of the Marker to compare.</td>
</tr>
<tr>
<td>onMatch</td>
<td>String</td>
<td>Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.</td>
</tr>
<tr>
<td>onMismatch</td>
<td>String</td>
<td>Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.</td>
</tr>
</tbody>
</table>

**Marker Filter Parameters**

A sample configuration that only allows the event to be written by the appender if the Marker matches:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <MarkerFilter marker="FLOW" onMatch="ACCEPT" onMismatch="DENY"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

11.1.6 RegexFilter

The RegexFilter allows the formatted or unformatted message to be compared against a regular expression.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex</td>
<td>String</td>
<td>The regular expression.</td>
</tr>
</tbody>
</table>
useRawMsg | **boolean** | If true the unformatted message will be used, otherwise the formatted message will be used. The default value is false.

onMatch | **String** | Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.

onMismatch | **String** | Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.

---

**Regex Filter Parameters**

A sample configuration that only allows the event to be written by the appender if it contains the word "test":

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
    <Appenders>
        <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
            <RegexpFilter regex=".* test .*" onMatch="ACCEPT" onMismatch="DENY"/>
            <PatternLayout>
                <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <TimeBasedTriggeringPolicy />
        </RollingFile>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="RollingFile"/>
        </Root>
    </Loggers>
</Configuration>
```

### 11.1.7 Script

The ScriptFilter executes a script that returns true or false.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>script</td>
<td>Script, ScriptFile or ScriptRef</td>
<td>The Script element that specifies the logic to be executed.</td>
</tr>
<tr>
<td>onMatch</td>
<td>String</td>
<td>Action to take when the script returns true. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.</td>
</tr>
</tbody>
</table>
### Script Filter Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration</td>
<td>Configuration</td>
<td>The Configuration that owns this ScriptFilter.</td>
</tr>
<tr>
<td>level</td>
<td>Level</td>
<td>The logging Level associated with the event. Only present when configured as a global filter.</td>
</tr>
<tr>
<td>loggerName</td>
<td>String</td>
<td>The name of the logger. Only present when configured as a global filter.</td>
</tr>
<tr>
<td>logEvent</td>
<td>LogEvent</td>
<td>The LogEvent being processed. Not present when configured as a global filter.</td>
</tr>
<tr>
<td>marker</td>
<td>Marker</td>
<td>The Marker passed on the logging call, if any. Only present when configured as a global filter.</td>
</tr>
<tr>
<td>message</td>
<td>Message</td>
<td>The Message associated with the logging call. Only present when configured as a global filter.</td>
</tr>
<tr>
<td>parameters</td>
<td>Object[]</td>
<td>The parameters passed to the logging call. Only present when configured as a global filter. Some Messages include the parameters as part of the Message.</td>
</tr>
<tr>
<td>throwable</td>
<td>Throwable</td>
<td>The Throwable passed to the logging call, if any. Only present when configured as a global filter. Some Messages include Throwable as part of the Message.</td>
</tr>
<tr>
<td>substitutor</td>
<td>StrSubstitutor</td>
<td>The StrSubstitutor used to replace lookup variables.</td>
</tr>
</tbody>
</table>

### Script Parameters

The sample below shows how to declare script fields and then reference them in specific components. See [ScriptCondition](#) for an example of how the `Script` element can be used to embed script code directly in the configuration.
11.1.8 StructuredDataFilter

The StructuredDataFilter is a MapFilter that also allows filtering on the event id, type and message.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyValuePair</td>
<td>KeyValuePair[]</td>
<td>One or more KeyValuePair elements that define the key in the map and the value to match on. &quot;id&quot;, &quot;id.name&quot;, &quot;type&quot;, and &quot;message&quot; should be used to match on the StructuredDataId, the name portion of the StructuredDataId, the type, and the formatted message respectively. If the same key is specified more than once then the check for that key will automatically be an &quot;or&quot; since a Map can only contain a single value.</td>
</tr>
</tbody>
</table>
### StructuredData Filter Parameters

As in this configuration, the StructuredDataFilter can be used to log particular events:

```xml
<Configuration status="warn" name="MyApp" packages="">
  <StructuredDataFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
    <KeyValuePair key="id" value="Login"/>
    <KeyValuePair key="id" value="Logout"/>
  </StructuredDataFilter>
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <BurstFilter level="INFO" rate="16" maxBurst="100"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

#### 11.1.9 ThreadContextMapFilter (or ContextMapFilter)

The ThreadContextMapFilter or ContextMapFilter allows filtering against data elements that are in the current context. By default this is the ThreadContext Map.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator</td>
<td>String</td>
<td>If the operator is &quot;or&quot; then a match by any one of the key/value pairs will be considered to be a match, otherwise all the key/value pairs must match.</td>
</tr>
<tr>
<td>onMatch</td>
<td>String</td>
<td>Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.</td>
</tr>
<tr>
<td>onMismatch</td>
<td>String</td>
<td>Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.</td>
</tr>
</tbody>
</table>
keyValuePair  KeyValuePair[]  One or more KeyValuePair elements that define the key in the map and the value to match on. If the same key is specified more than once then the check for that key will automatically be an "or" since a Map can only contain a single value.

operator  String  If the operator is "or" then a match by any one of the key/value pairs will be considered to be a match, otherwise all the key/value pairs must match.

onMatch  String  Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.

onMismatch  String  Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.

Context Map Filter Parameters

A configuration containing the ContextMapFilter might look like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <ContextMapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
    <KeyValuePair key="User1" value="DEBUG"/>
    <KeyValuePair key="User2" value="WARN"/>
  </ContextMapFilter>
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log" filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <BurstFilter level="INFO" rate="16" maxBurst="100"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
```

The ContextMapFilter can also be applied to a logger for filtering:
<xml version="1.0" encoding="UTF-8"/>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log"
      filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <BurstFilter level="INFO" rate="16" maxBurst="100"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
      <TimeBasedTriggeringPolicy />
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
      <ContextMapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
        <KeyValuePair key="foo" value="bar"/>
        <KeyValuePair key="User2" value="WARN"/>
      </ContextMapFilter>
    </Root>
  </Loggers>
</Configuration>

### 11.1.10 ThresholdFilter

This filter returns the onMatch result if the level in the LogEvent is the same or more specific than the configured level and the onMismatch value otherwise. For example, if the ThresholdFilter is configured with Level ERROR and the LogEvent contains Level DEBUG then the onMismatch value will be returned since ERROR events are more specific than DEBUG.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>String</td>
<td>A valid Level name to match on.</td>
</tr>
<tr>
<td>onMatch</td>
<td>String</td>
<td>Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.</td>
</tr>
<tr>
<td>onMismatch</td>
<td>String</td>
<td>Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.</td>
</tr>
</tbody>
</table>

**Threshold Filter Parameters**

A sample configuration that only allows the event to be written by the appender if the level matches:
11.1.11 TimeFilter

The time filter can be used to restrict filter to only a certain portion of the day.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>String</td>
<td>A time in HH:mm:ss format.</td>
</tr>
<tr>
<td>end</td>
<td>String</td>
<td>A time in HH:mm:ss format. Specifying an end time less than the start time will result in no log entries being written.</td>
</tr>
<tr>
<td>timezone</td>
<td>String</td>
<td>The timezone to use when comparing to the event timestamp.</td>
</tr>
<tr>
<td>onMatch</td>
<td>String</td>
<td>Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.</td>
</tr>
<tr>
<td>onMismatch</td>
<td>String</td>
<td>Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.</td>
</tr>
</tbody>
</table>

**Time Filter Parameters**

A sample configuration that only allows the event to be written by the appender from 5:00 to 5:30 am each day using the default timezone:
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <RollingFile name="RollingFile" fileName="logs/app.log"
      filePattern="logs/app-%d{MM-dd-yyyy}.log.gz">
      <TimeFilter start="05:00:00" end="05:30:00" onMatch="ACCEPT" onMismatch="DENY"/>
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
    </RollingFile>
    <TimeBasedTriggeringPolicy />
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingFile"/>
    </Root>
  </Loggers>
</Configuration>
12 Async Loggers

12.1 Asynchronous Loggers for Low-Latency Logging

Asynchronous logging can improve your application's performance by executing the I/O operations in a separate thread. Log4j 2 makes a number of improvements in this area.

- **Asynchronous Loggers** are a new addition in Log4j 2. Their aim is to return from the call to Logger.log to the application as soon as possible. You can choose between making all Loggers asynchronous or using a mixture of synchronous and asynchronous Loggers. Making all Loggers asynchronous will give the best performance, while mixing gives you more flexibility.

- **LMAX Disruptor technology.** Asynchronous Loggers internally use the Disruptor, a lock-free inter-thread communication library, instead of queues, resulting in higher throughput and lower latency.

- As part of the work for Async Loggers, **Asynchronous Appenders** have been enhanced to flush to disk at the end of a batch (when the queue is empty). This produces the same result as configuring "immediateFlush=true", that is, all received log events are always available on disk, but is more efficient because it does not need to touch the disk on each and every log event. (Async Appenders use ArrayBlockingQueue internally and do not need the disruptor jar on the classpath.)

12.1.1 Trade-offs

Although asynchronous logging can give significant performance benefits, there are situations where you may want to choose synchronous logging. This section describes some of the trade-offs of asynchronous logging.

**Benefits**

- Higher peak throughput. With an asynchronous logger your application can log messages at 6 - 68 times the rate of a synchronous logger.

  This is especially interesting for applications that occasionally need to log bursts of messages. Async logging can help prevent or dampen latency spikes by shortening the wait time until the next message can be logged. If the queue size is configured large enough to handle the burst, asynchronous logging will help prevent your application from falling behind (as much) during a sudden increase of activity.

- Lower logging response time latency. Response time latency is the time it takes for a call to Logger.log to return under a given workload. Asynchronous Loggers have consistently lower latency than synchronous loggers or even queue-based asynchronous appenders.

**Drawbacks**

- Error handling. If a problem happens during the logging process and an exception is thrown, it is less easy for an asynchronous logger or appender to signal this problem to the application. This can partly be alleviated by configuring an ExceptionHandler, but this may still not cover all cases. For this reason, if logging is part of your business logic, for example if you are using Log4j as an audit logging framework, we would recommend to synchronously log those audit messages. (Note that you can still combine them and use asynchronous logging for debug/trace logging in addition to synchronous logging for the audit trail.)

- In some rare cases, care must be taken with mutable messages. Most of the time you don't need to worry about this. Log4 will ensure that log messages like logger.debug("My object is ", myObject) will use the state of the myObject parameter at the time of the call to logger.debug(). The log message will not change even if myObject is modified later. It is safe to asynchronously log mutable objects because most Message implementations built-in to
Log4j take a snapshot of the parameters. There are some exceptions however: MapMessage and StructuredDataMessage are mutable by design: fields can be added to these messages after the message object was created. These messages should not be modified after they are logged with asynchronous loggers or asynchronous appenders; you may or may not see the modifications in the resulting log output. Similarly, custom Message implementations should be designed with asynchronous use in mind, and either take a snapshot of their parameters at construction time, or document their thread-safety characteristics.

- If your application is running in an environment where CPU resources are scarce, like a machine with one CPU with a single core, starting another thread is not likely to give better performance.
- If the sustained rate at which your application is logging messages is faster than the maximum sustained throughput of the underlying appender, the queue will fill up and the application will end up logging at the speed of the slowest appender. If this happens, consider selecting a faster appender, or logging less. If neither of these is an option, you may get better throughput and fewer latency spikes by logging synchronously.

### 12.1.2 Making All Loggers Asynchronous

Log4j-2.9 and higher require disruptor-3.3.4.jar or higher on the classpath. Prior to Log4j-2.9, disruptor-3.0.0.jar or higher was required.

This is simplest to configure and gives the best performance. To make all loggers asynchronous, add the disruptor jar to the classpath and set the system property log4j2.contextSelector to org.apache.logging.log4j.core.async.AsyncLoggerContextSelector.

By default, location is not passed to the I/O thread by asynchronous loggers. If one of your layouts or custom filters needs location information, you need to set "includeLocation=true" in the configuration of all relevant loggers, including the root logger.

A configuration that does not require location might look like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Don't forget to set system property
-Dlog4j2.contextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector
to make all loggers asynchronous. -->
<Configuration status="WARN">
    <Appenders>
        <!-- Async Loggers will auto-flush in batches, so switch off immediateFlush. -->
        <RandomAccessFile name="RandomAccessFile" fileName="async.log" immediateFlush="false" append="false">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m %ex%n</Pattern>
            </PatternLayout>
        </RandomAccessFile>
    </Appenders>
    <Loggers>
        <Root level="info" includeLocation="false">
            <AppenderRef ref="RandomAccessFile"/>
        </Root>
    </Loggers>
</Configuration>
```

When AsyncLoggerContextSelector is used to make all loggers asynchronous, make sure to use normal <root> and <logger> elements in the configuration. The AsyncLoggerContextSelector will
ensure that all loggers are asynchronous, using a mechanism that is different from what happens when you configure `<asyncRoot>` or `<asyncLogger>`. The latter elements are intended for mixing async with sync loggers. If you use both mechanisms together you will end up with two background threads, where your application passes the log message to thread A, which passes the message to thread B, which then finally logs the message to disk. This works, but there will be an unnecessary step in the middle.

There are a few system properties you can use to control aspects of the asynchronous logging subsystem. Some of these can be used to tune logging performance.

The below properties can also be specified by creating a file named `log4j2.component.properties` and including this file in the classpath of the application.

Note that system properties were renamed into a more consistent style in Log4j 2.10.0. All old property names are still supported which are documented [here](#).

<table>
<thead>
<tr>
<th>System Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log4j2.asyncLoggerExceptionHandler</code></td>
<td>default handler</td>
<td>Fully qualified name of a class that implements the <code>com.lmax.disruptor.ExceptionHandler</code> interface. The class needs to have a public zero-argument constructor. If specified, this class will be notified when an exception occurs while logging the messages. If not specified, the default exception handler will print a message and stack trace to the standard error output stream.</td>
</tr>
</tbody>
</table>
| `log4j2.asyncLoggerRingBufferSize` | `256 * 1024`  | Size (number of slots) in the RingBuffer used by the asynchronous logging subsystem. Make this value large enough to deal with bursts of activity. The minimum size is 128. The RingBuffer will be pre-allocated at first use and will never grow or shrink during the life of the system. When the application is logging faster than the underlying appender can keep up with for a long enough time to fill up the queue, the behaviour is determined by the `AsyncQueueFullPolicy`.
log4j2.asyncLoggerWaitStrategy Timeout

Valid values: Block, Timeout, Sleep, Yield.

Block is a strategy that uses a lock and condition variable for the I/O thread waiting for log events. Block can be used when throughput and low-latency are not as important as CPU resource. Recommended for resource constrained/virtualised environments.

Timeout is a variation of the Block strategy that will periodically wake up from the lock condition await() call. This ensures that if a notification is missed somehow the consumer thread is not stuck but will recover with a small latency delay (default 10ms).

Sleep is a strategy that initially spins, then uses a Thread.yield(), and eventually parks for the minimum number of nanos the OS and JVM will allow while the I/O thread is waiting for log events.

Sleep is a good compromise between performance and CPU resource. This strategy has very low impact on the application thread, in exchange for some additional latency for actually getting the message logged.

Yield is a strategy that uses a Thread.yield() for waiting for log events after an initially spinning.

Yield is a good compromise between performance and CPU resource, but may use more CPU than Sleep in order to get the message logged to disk sooner.

log4j2.asyncLoggerThreadNameStrategy CACHED

Valid values: CACHED, UNCACHED.

By default, AsyncLogger caches the thread name in a ThreadLocal variable to improve performance. Specify the UNCACHED option if your application modifies the thread name at runtime (with Thread.currentThread().setName()) and you want to see the new thread name reflected in the log.
Implementation of the org.apache.logging.log4j.core.util.Clock interface that is used for timestamping the log events when all loggers are asynchronous.

By default, System.currentTimeMillis is called on every log event.

CachedClock is an optimization intended for low-latency applications where time stamps are generated from a clock that updates its internal time in a background thread once every millisecond, or every 1024 log events, whichever comes first. This reduces logging latency a little, at the cost of some precision in the logged time stamps. Unless you are logging many events, you may see "jumps" of 10-16 milliseconds between log time stamps. WEB APPLICATION WARNING: The use of a background thread may cause issues for web applications and OSGi applications so CachedClock is not recommended for this kind of applications.

You can also specify the fully qualified class name of a custom class that implements the Clock interface.

System Properties to configure all asynchronous loggers

There are also a few system properties that can be used to maintain application throughput even when the underlying appender cannot keep up with the logging rate and the queue is filling up. See the details for system properties log4j2.asyncQueueFullPolicy and log4j2.discardThreshold.

12.1.3 Mixing Synchronous and Asynchronous Loggers

Log4j-2.9 and higher require disruptor-3.3.4.jar or higher on the classpath. Prior to Log4j-2.9, disruptor-3.0.0.jar or higher was required. There is no need to set system property "Log4jContextSelector" to any value.
Synchronous and asynchronous loggers can be combined in configuration. This gives you more flexibility at the cost of a slight loss in performance (compared to making all loggers asynchronous). Use the `<asyncRoot>` or `<asyncLogger>` configuration elements to specify the loggers that need to be asynchronous. A configuration can contain only one root logger (either a `<root>` or an `<asyncRoot>` element), but otherwise async and non-async loggers may be combined. For example, a configuration file containing `<asyncLogger>` elements can also contain `<root>` and `<logger>` elements for the synchronous loggers.

By default, location is not passed to the I/O thread by asynchronous loggers. If one of your layouts or custom filters needs location information, you need to set "includeLocation=true" in the configuration of all relevant loggers, including the root logger.

A configuration that mixes asynchronous loggers might look like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- No need to set system property "log4j2.contextSelector" to any value when using <asyncLogger> or <asyncRoot>. -->
<Configuration status="WARN">
    <Appenders>
        <!-- Async Loggers will auto-flush in batches, so switch off immediateFlush. -->
        <RandomAccessFile name="RandomAccessFile" fileName="asyncWithLocation.log"
            immediateFlush="false" append="false">
            <PatternLayout>
                <Pattern>%d [%t] %location %m %ex%n</Pattern>
            </PatternLayout>
        </RandomAccessFile>
    </Appenders>
    <Loggers>
        <!-- pattern layout actually uses location, so we need to include it -->
        <AsyncLogger name="com.foo.Bar" level="trace" includeLocation="true">
            <AppenderRef ref="RandomAccessFile"/>
        </AsyncLogger>
        <Root level="info" includeLocation="true">
            <AppenderRef ref="RandomAccessFile"/>
        </Root>
    </Loggers>
</Configuration>
```

There are a few system properties you can use to control aspects of the asynchronous logging subsystem. Some of these can be used to tune logging performance.

The below properties can also be specified by creating a file named `log4j2.component.properties` and including this file in the classpath of the application.

Note that system properties were renamed into a more consistent style in Log4j 2.10. All old property names are still supported which are documented here.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log4j2.asyncLoggerConfigExceptionHandler</td>
<td>Fully qualified name of a class that implements the com.lmax.disruptor.ExceptionHandler interface. The class needs to have a public zero-argument constructor. If specified, this class will be notified when an exception occurs while logging the messages. If not specified, the default exception handler will print a message and stack trace to the standard error output stream.</td>
</tr>
<tr>
<td>log4j2.asyncLoggerConfigRingBufferSize</td>
<td>Size (number of slots) in the RingBuffer used by the asynchronous logging subsystem. Make this value large enough to deal with bursts of activity. The minimum size is 128. The RingBuffer will be pre-allocated at first use and will never grow or shrink during the life of the system. When the application is logging faster than the underlying appender can keep up with for a long enough time to fill up the queue, the behaviour is determined by the AsyncQueueFullPolicy.</td>
</tr>
</tbody>
</table>
log4j2.asyncLoggerConfigWaitStrategy

Valid values: Block, Timeout, Sleep, Yield.

Block is a strategy that uses a lock and condition variable for the I/O thread waiting for log events. Block can be used when throughput and low-latency are not as important as CPU resource. Recommended for resource constrained/virtualised environments.

Timeout is a variation of the Block strategy that will periodically wake up from the lock condition await() call. This ensures that if a notification is missed somehow the consumer thread is not stuck but will recover with a small latency delay (default 10ms).

Sleep is a strategy that initially spins, then uses a Thread.yield(), and eventually parks for the minimum number of nanos the OS and JVM will allow while the I/O thread is waiting for log events. Sleep is a good compromise between performance and CPU resource. This strategy has very low impact on the application thread, in exchange for some additional latency for actually getting the message logged.

Yield is a strategy that uses a Thread.yield() for waiting for log events after an initially spinning. Yield is a good compromise between performance and CPU resource, but may use more CPU than Sleep in order to get the message logged to disk sooner.

System Properties to configure mixed asynchronous and normal loggers

There are also a few system properties that can be used to maintain application throughput even when the underlying appender cannot keep up with the logging rate and the queue is filling up. See the details for system properties log4j2.asyncQueueFullPolicy and log4j2.discardThreshold.

12.1.4 Location, location, location...

If one of the layouts is configured with a location-related attribute like HTML. locationInfo, or one of the patterns %C or $class, %F or %file, %l or %location, %L or %line, %M or %method, Log4j will take a snapshot of the stack, and walk the stack trace to find the location information.
This is an expensive operation: 1.3 - 5 times slower for synchronous loggers. Synchronous loggers wait as long as possible before they take this stack snapshot. If no location is required, the snapshot will never be taken.

However, asynchronous loggers need to make this decision before passing the log message to another thread; the location information will be lost after that point. The performance impact of taking a stack trace snapshot is even higher for asynchronous loggers: logging with location is 30-100 times slower than without location. For this reason, asynchronous loggers and asynchronous appenders do not include location information by default.

You can override the default behaviour in your logger or asynchronous appender configuration by specifying `includeLocation="true"`.

### 12.1.5 Asynchronous Logging Performance

The throughput performance results below were derived from running the PerfTest, MTPerfTest and PerfTestDriver classes which can be found in the Log4j 2 unit test source directory. For throughput tests, the methodology used was:

- First, warm up the JVM by logging 200,000 log messages of 500 characters.
- Repeat the warm-up 10 times, then wait 10 seconds for the I/O thread to catch up and buffers to drain.
- Measure how long it takes to execute 256 * 1024 / threadCount calls to Logger.log and express the result in messages per second.
- Repeat the test 5 times and average the results.

The results below were obtained with log4j-2.0-beta5, disruptor-3.0.0.beta3, log4j-1.2.17 and logback-1.0.10.

#### 12.1.5.1 Logging Peak Throughput

The graph below compares the throughput of synchronous loggers, asynchronous appenders and asynchronous loggers. This is the total throughput of all threads together. In the test with 64 threads, asynchronous loggers are 12 times faster than asynchronous appenders, and 68 times faster than synchronous loggers.

Asynchronous loggers’ throughput increases with the number of threads, whereas both synchronous loggers and asynchronous appenders have more or less constant throughput regardless of the number of threads that are doing the logging.
12.1.5.2 Asynchronous Throughput Comparison with Other Logging Packages

We also compared peak throughput of asynchronous loggers to the synchronous loggers and asynchronous appenders available in other logging packages, specifically log4j-1.2.17 and logback-1.0.10, with similar results. For asynchronous appenders, total logging throughput of all threads together remains roughly constant when adding more threads. Asynchronous loggers make more effective use of the multiple cores available on the machine in multi-threaded scenarios.

On Solaris 10 (64bit) with JDK1.7.0_06, 4-core Xeon X5570 dual CPU @2.93Ghz with hyperthreading switched on (16 virtual cores):
Throughput per thread in messages/second

On Windows 7 (64bit) with JDK1.7.0_11, 2-core Intel i5-3317u CPU @ 1.70Ghz with hyperthreading switched on (4 virtual cores):

<table>
<thead>
<tr>
<th>Logger</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
<th>64 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log4j 2: Loggers all asynchronous</td>
<td>2,652,412</td>
<td>909,119</td>
<td>776,993</td>
<td>516,365</td>
<td>239,246</td>
<td>253,791</td>
<td>288,997</td>
</tr>
<tr>
<td>Log4j 2: Loggers mixed sync/async</td>
<td>2,454,358</td>
<td>839,394</td>
<td>854,578</td>
<td>597,913</td>
<td>261,003</td>
<td>216,863</td>
<td>218,937</td>
</tr>
<tr>
<td>Log4j 2: Async Appender</td>
<td>1,713,429</td>
<td>603,019</td>
<td>331,506</td>
<td>149,408</td>
<td>86,107</td>
<td>45,529</td>
<td>23,980</td>
</tr>
<tr>
<td>Log4j1: Async Appender</td>
<td>2,239,664</td>
<td>494,470</td>
<td>221,402</td>
<td>109,314</td>
<td>60,580</td>
<td>31,706</td>
<td>14,072</td>
</tr>
<tr>
<td>Logback: Async Appender</td>
<td>2,206,907</td>
<td>624,082</td>
<td>307,500</td>
<td>160,096</td>
<td>85,701</td>
<td>43,422</td>
<td>21,303</td>
</tr>
<tr>
<td>Log4j 2: Synchronous</td>
<td>273,536</td>
<td>136,523</td>
<td>67,609</td>
<td>34,404</td>
<td>15,373</td>
<td>7,903</td>
<td>4,253</td>
</tr>
<tr>
<td>Log4j1: Synchronous</td>
<td>326,894</td>
<td>105,591</td>
<td>57,036</td>
<td>30,511</td>
<td>13,900</td>
<td>7,094</td>
<td>3,509</td>
</tr>
<tr>
<td>Logback: Synchronous</td>
<td>178,063</td>
<td>65,000</td>
<td>34,372</td>
<td>16,903</td>
<td>8,334</td>
<td>3,985</td>
<td>1,967</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logger</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log4j 2: Loggers all asynchronous</td>
<td>1,715,344</td>
<td>928,951</td>
<td>1,045,265</td>
<td>1,509,109</td>
<td>1,708,989</td>
<td>773,565</td>
</tr>
<tr>
<td>Log4j 2: Loggers mixed sync/async</td>
<td>571,099</td>
<td>1,204,774</td>
<td>1,632,204</td>
<td>1,368,041</td>
<td>462,093</td>
<td>908,529</td>
</tr>
<tr>
<td>Log4j 2: Async Appender</td>
<td>1,236,548</td>
<td>1,006,287</td>
<td>511,571</td>
<td>302,230</td>
<td>160,094</td>
<td>60,152</td>
</tr>
<tr>
<td>Log4j1: Async Appender</td>
<td>1,373,195</td>
<td>911,657</td>
<td>636,899</td>
<td>406,405</td>
<td>202,777</td>
<td>162,964</td>
</tr>
<tr>
<td>Logback: Async Appender</td>
<td>1,979,515</td>
<td>783,722</td>
<td>582,935</td>
<td>289,905</td>
<td>172,463</td>
<td>133,435</td>
</tr>
<tr>
<td>Log4j 2: Synchronous</td>
<td>281,250</td>
<td>225,731</td>
<td>129,015</td>
<td>66,590</td>
<td>34,401</td>
<td>17,347</td>
</tr>
</tbody>
</table>
Log4j1: Synchronous
147,824 72,383 32,865 18,025 8,937 4,440
Logback: Synchronous
149,811 66,301 32,341 16,962 8,431 3,610

Throughput per thread in messages/second

12.1.5.3 Response Time Latency

This section has been rewritten with the Log4j 2.6 release. The previous version only reported service time instead of response time. See the response time side bar on the performance page on why this is too optimistic. Furthermore the previous version reported average latency, which does not make sense since latency is not a normal distribution. Finally, the previous version of this section only reported the maximum latency of up to 99.99% of the measurements, which does not tell you how bad the worst 0.01% were. This is unfortunate because often the "outliers" are all that matter when it comes to response time. From this release we will try to do better and report response time latency across the full range of percentages, including all the outliers. Our thanks to Gil Tene for his How NOT to measure latency presentation. (Now we know why this is also known as the "Oh s#@t!" presentation.)

Response time is how long it takes to log a message under a certain load. What is often reported as latency is actually service time; how long it took to perform the operation. This hides the fact that a single spike in service time adds queueing delay for many of the subsequent operations. Service time is easy to measure (and often looks good on paper) but is irrelevant for users since it omits the time spent waiting for service. For this reason we report response time: service time plus wait time.

The response time test results below were all derived from running the ResponseTimeTest class which can be found in the Log4j 2 unit test source directory. If you want to run these tests yourself, here are the command line options we used:

-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationConcurrentTime -XX:+PrintGCApplicationStoppedTime (to eyeball GC and safepoint pauses)

The graph below compares response time latency of the ArrayBlockingQueue-based asynchronous appenders in Logback 1.1.7, Log4j 1.2.17 to the various options for asynchronous logging that Log4j 2.6 offers. Under a workload of 128,000 messages per second, using 16 threads (each logging at a rate of 8,000 messages per second), we see that Logback 1.1.7, Log4j 1.2.17 experience latency spikes that are orders of magnitude larger than Log4j 2.
The graph below zooms in on the Log4j 2 results for the same test. We see that the worst-case response time is highest for the ArrayBlockingQueue-based Async Appender. Garbage-free async loggers have the best response time behaviour.
12.1.6 Under The Hood

Asynchronous Loggers are implemented using the LMAX Disruptor inter-thread messaging library. From the LMAX web site:

... using queues to pass data between stages of the system was introducing latency, so we focused on optimising this area. The Disruptor is the result of our research and testing. We found that cache misses at the CPU-level, and locks requiring kernel arbitration are both extremely costly, so we created a framework which has "mechanical sympathy" for the hardware it's running on, and that's lock-free.

LMAX Disruptor internal performance comparisons with java.util.concurrent.ArrayBlockingQueue can be found here.
13 Garbage-free Logging

13.1 Garbage-free Steady State Logging

Garbage collection pauses are a common cause of latency spikes and for many systems significant effort is spent on controlling these pauses.

Many logging libraries, including previous versions of Log4j, allocate temporary objects like log event objects, Strings, char arrays, byte arrays and more during steady state logging. This contributes to pressure on the garbage collector and increases the frequency with which GC pauses occur.

From version 2.6, Log4j runs in "garbage free" mode by default where objects and buffers are reused and no temporary objects are allocated as much as possible. There is also a "low garbage" mode which is not completely garbage free but does not use ThreadLocal fields. This is the default mode when Log4j detects it is running in a web application. Finally, it is possible to switch off all garbage-free logic and run in "classic mode" instead. For details, see the Configuration section below.

13.1.1 A Contrived Example

To highlight the difference that garbage-free logging can make, we used Java Flight Recorder to measure a simple application that does nothing but log a simple string as often as possible for about 12 seconds.

The application was configured to use Async Loggers, a RandomAccessFile appender and a "%d %p %c{1.} [%t] %m %ex%n" pattern layout. (Async Loggers used the Yield WaitStrategy.)

Mission Control shows that with Log4j 2.5 this application allocates memory at a rate of about 809 MB/sec, resulting in 141 minor collections. Log4j 2.6 does not allocate temporary objects in this configuration, and as a result the same application with Log4j 2.6 has a memory allocation rate of 1.6 MB/sec and was GC-free with 0 (zero) garbage collections.

13.1.2 Configuration

Garbage-free logging in Log4j 2.6 is partially implemented by reusing objects in ThreadLocal fields, and partially by reusing buffers when converting text to bytes.
ThreadLocal fields holding non-JDK classes can cause memory leaks in web applications when the application server's thread pool continues to reference these fields after the web application is undeployed. To avoid causing memory leaks, Log4j will not use these ThreadLocals when it detects that it is used in a web application (when the `javax.servlet.Servlet` class is in the classpath, or when system property `log4j2.isWebapp` is set to "true").

Some garbage-reducing functionality does not rely on ThreadLocals and is enabled by default for all applications: in Log4j 2.6, converting log events to text and text to bytes can be done by directly encoding text into a reused ByteBuffer without creating intermediary Strings, char arrays and byte arrays. So while logging is not completely garbage-free for web applications yet, the pressure on the garbage collector can still be significantly reduced.

**Note 1:** as of version 2.6, a Log4j configuration containing a `<Properties>` section will result in temporary objects being created during steady-state logging.

**Note 2:** the Async Logger Timeout wait strategy (the default) and the Block wait strategy cause `java.util.concurrent.locks.AbstractQueuedSynchronizer$Node` objects to be created. The Yield and Sleep wait strategies are garbage-free. (See [here](#) and [here](#).)

### 13.1.2.1 Disabling Garbage-free Logging

There are two separate system properties for manually controlling the mechanisms Log4j uses to avoid creating temporary objects:

- `log4j2.enableThreadlocals` - if "true" (the default for non-web applications) objects are stored in ThreadLocal fields and reused, otherwise new objects are created for each log event.
- `log4j2.enableDirectEncoders` - if "true" (the default) log events are converted to text and this text is converted to bytes without creating temporary objects. Note: synchronous logging performance may be worse for multi-threaded applications in this mode due to synchronization on the shared buffer. If your application is multi-threaded and logging performance is important, consider using Async Loggers.
- The ThreadContext map is not garbage-free by default, but from Log4j 2.7 it can be configured to be garbage-free by setting system property `log4j2.garbagefreeThreadContextMap` to "true".

Instead of system properties, the above properties can also be specified in a file named `log4j2.component.properties` by including this file in the classpath of the application. See the manual regarding system properties for more info.

### 13.1.2.2 Supported Appenders

The following appenders are garbage-free during steady-state logging:

- Console
- File
- RollingFile (some temporary objects are created during file rollover)
- RandomAccessFile
- RollingRandomAccessFile (some temporary objects are created during file rollover)
- MemoryMappedFile

Any other appenders not in the above list (including AsyncAppender) create temporary objects during steady-state logging. Instead of AsyncAppender, use Async Loggers to log asynchronously in a garbage-free manner.
13.1.2.3 Supported Filters

The following filters are garbage-free during steady-state logging:

- CompositeFilter (adding and removing element filters creates temporary objects for thread safety)
- DynamicThresholdFilter
- LevelRangeFilter (garbage free since 2.8)
- MapFilter (garbage free since 2.8)
- MarkerFilter (garbage free since 2.8)
- StructuredDataFilter (garbage free since 2.8)
- ThreadContextMapFilter (garbage free since 2.8)
- ThresholdFilter (garbage free since 2.8)
- TimeFilter (garbage free since 2.8)

Other filters like BurstFilter, RegexFilter and ScriptFilter are not trivial to make garbage free, and there is currently no plan to change them.

13.1.2.4 Supported Layouts

13.GelfLayout

GelfLayout is garbage-free when used with compressionType="OFF", as long as no additional field contains '${' (variable substitution).

13.PatternLayout

PatternLayout with the following limited set of conversion patterns is garbage-free. Format modifiers to control such things as field width, padding, left and right justification will not generate garbage.

<table>
<thead>
<tr>
<th>Conversion Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c{precision}, %logger{precision}</td>
<td>Logger name</td>
</tr>
</tbody>
</table>
%d, %date

Note: Only the predefined date formats are garbage-free: (millisecond separator may be either a comma ',' or a period '.')

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d(DEFAULT)</td>
<td>2012-11-02 14:34:02,781</td>
</tr>
<tr>
<td>%d(ISO8601)</td>
<td>2012-11-02T14:34:02,781</td>
</tr>
<tr>
<td>%d(ISO8601_BASIC)</td>
<td>20121102T143402,781</td>
</tr>
<tr>
<td>%d(Absolute)</td>
<td>14:34:02,781</td>
</tr>
<tr>
<td>%d(DATE)</td>
<td>02 Nov 2012 14:34:02,781</td>
</tr>
<tr>
<td>%d(COMPACT)</td>
<td>20121102143402781</td>
</tr>
<tr>
<td>%d(HH:mm:ss,SSS)</td>
<td>14:34:02,781</td>
</tr>
<tr>
<td>%d(dd MMM yyyy)</td>
<td>02 Nov 2012 14:34:02,781</td>
</tr>
<tr>
<td>%d(HH:mm:ss)(GMT+0)</td>
<td>18:34:02</td>
</tr>
<tr>
<td>%d(UNIX)</td>
<td>1351866842</td>
</tr>
<tr>
<td>%d(UNIX_MILLIS)</td>
<td>1351866842781</td>
</tr>
</tbody>
</table>

%enc(pattern), %encode(pattern)

Encodes special characters such as 'n' and HTML characters to help prevent log forging and some XSS attacks that could occur when displaying logs in a web browser - garbage-free since 2.8

%equals(pattern){test}{substitution}, %equalsIgnoreCase(pattern){test}{substitution}

Replaces occurrences of 'test', a string, with its replacement 'substitution' in the string resulting from evaluation of the pattern - garbage-free since 2.8

%highlight(pattern){style}

Adds ANSI colors - garbage-free since 2.7 (unless nested pattern is not garbage free)

K(key), map(key), MAP(key)

Outputs the entries in a MapMessage, if one is present in the event - garbage-free since 2.8.

%m, %msg, %message

Log message (garbage-free unless message text contains '${'}

%marker

The full name of the marker (including parents) - garbage-free since 2.8

%markerSimpleName

The simple name of the marker (not including parents)

%maxLen, %maxLength

Truncates another pattern to some max number of characters - garbage-free since 2.8

%n

The platform dependent line separator

%n, %nano

System.nanoTime() when the event was logged

%notEmpty(pattern), %varsNotEmpty(pattern), %variablesNotEmpty(pattern)

Outputs the result of evaluating the pattern if and only if all variables in the pattern are not empty - garbage-free since 2.8

%p, %level

The level of the logging event

%r, %relative

The number of milliseconds elapsed since the JVM was started until the creation of the logging event - garbage-free since 2.8
Other PatternLayout conversion patterns, and other Layouts may be updated to avoid creating temporary objects in future releases. (Patches welcome!)

**Note:** Logging exceptions and stack traces will create temporary objects with any layout. (However, Layouts will only create these temporary objects when an exception actually occurs.) We haven’t figured out a way to log exceptions and stack traces without creating temporary objects. That is unfortunate, but you probably still want to log them when they happen.

**Note:** patterns containing regular expressions and lookups for property substitution will result in temporary objects being created during steady-state logging.

Including location information is done by walking the stacktrace of an exception, which creates temporary objects, so the following patterns are not garbage-free:

- `%C, %class - Class Name`
- `%F, %file - File Location`
- `%l, %location - Location`
- `%L, %line - Line Location`
- `%M, %method - Method Location`

Also, the pattern converters for formatting Throwables are not garbage-free:

- `%ex, %exception, %Throwable - The Throwable trace bound to the LoggingEvent`
- `%rEx, %rException %rThrowable - Same as %ex but with wrapping exceptions`
- `%xEx, %xException, %xThrowable - Same as %ex but with class packaging information`
- `%u, %uuid - Creates a new random or time-based UUID while formatting`

### 13.1.2.5 API Changes

Methods have been added to the `Logger` interface so that no vararg array objects are created when logging messages with up to ten parameters.

Also, methods have been added to the `Logger` interface to log `java.lang.CharSequence` messages. User-defined objects that implement the `CharSequence` interface can be logged without creating temporary objects: Log4j will try to turn `CharSequence` messages, `Object` messages and
message parameters into text by appending them to a StringBuilder as a CharSequence. This avoids calling toString() on these objects.

An alternative is to implement the org.apache.logging.log4j.util.StringBuilderFormattable interface. If an object is logged that implements this interface, its formatTo method is called instead of toString().

Log4j may call toString() on message and parameter objects when garbage-free logging is disabled (when system property log4j2.enableThreadlocals is set to "false").

13.1.2.6 Impact on Application Code: Autoboxing

We made an effort to make logging garbage-free without requiring code changes in existing applications, but there is one area where this was not possible. When logging primitive values (i.e. int, double, boolean, etc.) the JVM autoboxes these primitive values to their Object wrapper equivalents, creating garbage.

Log4j provides an Unbox utility to prevent autoboxing of primitive parameters. This utility contains a thread-local pool of reused StringBuilders. The Unbox.box(primitive) methods write directly into a StringBuilder, and the resulting text will be copied into the final log message text without creating temporary objects.

    import static org.apache.logging.log4j.util.Unbox.box;
    ...
    public void garbageFree() {
      logger.debug("Prevent primitive autoboxing {} {}", box(10L), box(2.6d));
    }

Note: not all logging is garbage free. Specifically:

- The ThreadContext map is not garbage-free by default, but can be configured to be garbage-free by setting system property log4j2.garbagefreeThreadContextMap to "true".
- The ThreadContext stack is not garbage-free.
- Logging more than 10 parameters creates vararg arrays.
- Logging very large messages (more than 518 characters) when all loggers are Async Loggers will cause the internal StringBuilder in the RingBuffer to be trimmed back to their max size.
- Logging messages containing '${': substituting a ${variable} creates temporary objects.
- Logging a lambda as a parameter logger.info("lambda value is {}", () -> callExpensiveMethod()) creates a vararg array. Logging a lambda expression by itself is garbage-free: logger.debug(() -> callExpensiveMethod()).
- The Logger.traceEntry and Logger.traceExit methods create temporary objects.

13.1.3 Performance

13.1.3.1 Response Time Latency

Response time is how long it takes to log a message under a certain load. What is often reported as latency is actually service time: how long it took to perform the operation. This hides the fact that a
single spike in service time adds queueing delay for many of the subsequent operations. Service time is easy to measure (and often looks good on paper) but is irrelevant for users since it omits the time spent waiting for service. For this reason we report response time: service time plus wait time. See the response time section of the performance page for more detail.

The response time test results below were all derived from running the ResponseTimeTest class which can be found in the Log4j 2 unit test source directory. If you want to run these tests yourself, here are the command line options we used:

- `-Xms1G -Xmx1G` (prevent heap resizing during the test)

- **classic mode**: `-Dlog4j2.enable.threadlocals=false -Dlog4j2.enable.direct.encoders=false`

- **garbage-free mode**: `-Dlog4j2.enable.threadlocals=true -Dlog4j2.enable.direct.encoders=true`

- `-XX:CompileCommand=dontinline,org.apache.logging.log4j.core.async.perftest.NoOpIdleStrategy::idle`
- `-verbose:gc -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCDetails -XX:+PrintGCApplicationConcurrentTime -XX:+PrintGCApplicationStoppedTime` (to eyeball GC and safepoint pauses)

13.1.3.2 Async Loggers

The graph below compares "classic" logging to garbage-free logging response time behaviour for Log4j's Async Loggers. In the graph, "100k," means logging at a sustained load of 100,000 messages/second, "800k" is a sustained load of 800,000 messages/second.

In **classic** mode we see numerous minor garbage collections which pause the application threads for 3 milliseconds or more. This quickly adds up to response time delays of almost 10 milliseconds. As you can see in the graph, increasing the load shifts the curve to the left (there are more spikes). This makes sense: logging more means more pressure on the garbage collector resulting in more minor GC
pauses. We experimented a little with reducing the load to 50,000 or even 5000 messages/second, but this did not eliminate the 3 millisecond pauses, it just made them occur less frequently. Note that all GC pauses in this test are minor GC pauses. We did not see any full garbage collections.

In garbage-free mode, maximum response time remains well below 1 millisecond under a wide range of loads. (Max 780 us at 800,000 messages/sec, max 407 us at 600,000 messages/sec, with the 99% around 5 us for all loads up to 800,000 messages/sec.) Increasing or decreasing the load does not change the response time behaviour. We did not investigate the cause of the 200-300 microsecond pauses we saw in these tests.

When we increased the load further we begin to see larger response time pauses for both classic and garbage-free logging. At sustained loads of 1 million messages/second or more we start to approach the maximum throughput of the underlying RandomAccessFile Appender (see the synchronous logging throughput chart below). At these loads the ringbuffer starts to fill up and backpressure kicks in: attempting to add another message when the ringbuffer is full will block until a free slot becomes available. We start to see response times of tens of milliseconds or more; and attempting to increase the load even more results in larger and larger response time spikes.

13.1.3.3 Synchronous File Logging

With synchronous file logging, garbage-free logging still performs better than classic logging, but the difference is less pronounced.

At a workload of 100,000 messages/second, classic logging max response time was a little over 2 milliseconds where garbage-free logging was a little over 1 millisecond. When the workload is increased to 300,000 messages/second, classic logging shows response time pauses of 6 milliseconds where the garbage-free response times were less than 3 milliseconds. It may be possible to improve on this, we did not investigate further yet.

The above results are obtained with the ResponseTimeTest class which can be found in the Log4j 2 unit test source directory, running on JDK 1.8.0_45 on RHEL 6.5 (Linux 2.6.32-573.1.1.el6.x86_64) with 10-core Xeon CPU E5-2660 v3 @2.60GHz with hyperthreading switched on (20 virtual cores).
13.1.3.4 Classic Logging has Slightly Higher Throughput

Throughput is slightly worse for garbage-free logging, compared to classic logging. This is true for both synchronous and asynchronous logging. The graph below compares the sustained throughput of synchronous logging to a file with Log4j 2.6 in garbage-free mode, classic mode and Log4j 2.5.

The results above are obtained with the JMH Java benchmark harness. See the FileAppenderBenchmark source code in the log4j-perf module.

13.1.4 Under the Hood

Custom Message implementations that implement org.apache.logging.log4j.util.StringBuilderFormattable can be converted to text by garbage-free Layouts without creating temporary objects. PatternLayout uses this mechanism and other layouts that convert LogEvents to text will likely also look for this interface.

Custom Layouts that want to be garbage-free should implement the Encoder<LogEvent> interface. For custom Layouts that convert a LogEvent to a text representation, the org.apache.logging.log4j.core.layout.StringBuilderEncoder class may be useful to convert this text to bytes in a garbage-free manner.

Custom Appenders that want to be garbage-free should provide their Layout with a ByteBufferDestination implementation that the Layout can directly write into.

AbstractOutputStreamAppender has been modified to make the ConsoleAppender, (Rolling)FileAppender, (Rolling)RandomAccessFileAppender and MemoryMappedFileAppender garbage-free. An effort has been made to minimize impact on custom Appenders that extend AbstractOutputStreamAppender, but it is impossible to guarantee that changing the superclass will not impact any and all subclasses. Custom Appenders that extend AbstractOutputStreamAppender should verify that they still function correctly. In case there is a problem, system property log4j2.enable.direct.encoders can be set to "false" to revert to the pre-Log4j 2.6 behaviour.
14 JMX

14.1 JMX

Log4j 2 has built-in support for JMX. The StatusLogger, ContextSelector, and all LoggerContexts, LoggerConfigs and Appenders are instrumented with MBeans and can be remotely monitored and controlled.

Also included is a simple client GUI that can be used to monitor the StatusLogger output, as well as to remotely reconfigure Log4j with a different configuration file, or to edit the current configuration directly.

14.2 Enabling JMX

JMX support is enabled by default. When Log4j initializes, the StatusLogger, ContextSelector, and all LoggerContexts, LoggerConfigs and Appenders are instrumented with MBeans. To disable JMX completely, and prevent these MBeans from being created, specify system property `log4j2.disableJmx` to `true` when you start the Java VM.

14.2.1 Local Monitoring and Management

To perform local monitoring you don't need to specify any system properties. The JConsole tool that is included in the Java JDK can be used to monitor your application. Start JConsole by typing `$JAVA_HOME/bin/jconsole` in a command shell. For more details, see Oracle's documentation on how to use JConsole.

14.2.2 Remote Monitoring and Management

To enable monitoring and management from remote systems, set the following system property when starting the Java VM.

```
com.sun.management.jmxremote.port=portNum
```

In the property above, `portNum` is the port number through which you want to enable JMX RMI connections.

For more details, see Oracle's documentation on Remote Monitoring and Management.

14.2.3 RMI impact on Garbage Collection

Be aware that RMI by default triggers a full GC every hour. See the Oracle documentation for the `sun.rmi.dgc.server.gcInterval` and `sun.rmi.dgc.client.gcInterval` properties. The default value of both properties is 3600000 milliseconds (one hour). Before Java 6, it was one minute.

The two `sun.rmi` arguments reflect whether your JVM is running in server or client mode. If you want to modify the GC interval time it may be best to specify both properties to ensure the argument is picked up by the JVM.

An alternative may be to disable explicit calls to `System.gc()` altogether with `XX:+DisableExplicitGC`, or (if you are using the CMS or G1 collector) add `-XX:`
+ExplicitGCInvokesConcurrent to ensure the full GCs are done concurrently in parallel with your application instead of forcing a stop-the-world collection.

### 14.3 Log4j Instrumented Components

The best way to find out which methods and attributes of the various Log4j components are accessible via JMX is to look at the [Javadoc](https://logging.apache.org/log4j/1.2/faq.html) or by exploring directly in JConsole.

The screenshot below shows the Log4j MBeans in JConsole.

![Log4j MBeans in JConsole](image)

### 14.4 Client GUI

Log4j includes a basic client GUI that can be used to monitor the StatusLogger output and to remotely modify the Log4j configuration. The client GUI can be run as a stand-alone application or as a JConsole plug-in.

#### 14.4.1 Running the Client GUI as a JConsole Plug-in

To run the Log4j JMX Client GUI as a JConsole Plug-in, start JConsole with the following command:

or on Windows:

%JAVA_HOME%/bin/jconsole -pluginpath \path\to\log4j-api-${Log4jReleaseVersion}.jar;\path\to\log4j-core-${Log4jReleaseVersion}.jar;\path\to\log4j-jmx-gui-${Log4jReleaseVersion}.jar

If you execute the above command and connect to your application, you will see an extra "Log4j 2" tab in the JConsole window. This tab contains the client GUI, with the StatusLogger selected. The screenshot below shows the StatusLogger panel in JConsole.

14.4.2 Remotely Editing the Log4j Configuration

The client GUI also contains a simple editor that can be used to remotely change the Log4j configuration.

The screenshot below shows the configuration edit panel in JConsole.
The configuration edit panel provides two ways to modify the Log4j configuration: specifying a different configuration location URI, or modifying the configuration XML directly in the editor panel.

If you specify a different configuration location URI and click the "Reconfigure from Location" button, the specified file or resource must exist and be readable by the application, or an error will occur and the configuration will not change. If an error occurred while processing the contents of the specified resource, Log4j will keep its original configuration, but the editor panel will show the contents of the file you specified.

The text area showing the contents of the configuration file is editable, and you can directly modify the configuration in this editor panel. Clicking the "Reconfigure with XML below" button will send the configuration text to the remote application where it will be used to reconfigure Log4j on the fly. This will not overwrite any configuration file. Reconfiguring with text from the editor happens in memory only and the text is not permanently stored anywhere.

### 14.4.3 Running the Client GUI as a Stand-alone Application

To run the Log4j JMX Client GUI as a stand-alone application, run the following command:

or on Windows:
%JAVA_HOME%\bin\java -cp \path\to\log4j-api-${Log4jReleaseVersion}.jar;\path\to\log4j-core-${Log4jReleaseVersion}.jar;\path\to\log4j-jmx-gui-${Log4jReleaseVersion}.jar org.apache.logging.log4j.jmx.gui.ClientGui <options>

Where options are one of the following:

- `<host>[:<port>]`
- `service:jmx:rmi:///jndi/rmi://<host>:<port>/jmxrmi`
- `service:jmx:rmi://<host>::<port>/jndi/rmi://<host>:<port>/jmxrmi`

The port number must be the same as the portNum specified when you started the application you want to monitor.

For example, if you started your application with these options:

```
com.sun.management.jmxremote.port=33445
com.sun.management.jmxremote.authenticate=false
com.sun.management.jmxremote.ssl=false
```

(Note that this disables all security so this is not recommended for production environments. Oracle's documentation on Remote Monitoring and Management provides details on how to configure JMX more securely with password authentication and SSL.)

Then you can run the client with this command:


or on Windows:
%JAVA_HOME%\bin\java -cp \path\to\log4j-api-${Log4jReleaseVersion}.jar;\path\to\log4j-core-${Log4jReleaseVersion}.jar;\path\to\log4j-jmx-gui-${Log4jReleaseVersion}.jar org.apache.logging.log4j.jmx.gui.ClientGui localhost:33445

The screenshot below shows the StatusLogger panel of the client GUI when running as a stand-alone application.
The screenshot below shows the configuration editor panel of the client GUI when running as a stand-alone application.
15 Logging Separation

15.1 Logging Separation

There are many well known use cases where applications may share an environment with other applications and each has a need to have its own, separate logging environment. This purpose of this section is to discuss some of these cases and ways to accomplish this.

15.1.1 Use Cases

This section describes some of the use cases where Log4j could be used and what its desired behavior might be.

15.1.1.1 Standalone Application

Standalone applications are usually relatively simple. They typically have one bundled executable that requires only a single logging configuration.

15.1.1.2 Web Applications

A typical web application will be packaged as a WAR file and will include all of its dependencies in WEB-INF/lib and will have its configuration file located in the class path or in a location configured in the web.xml. Be sure to follow the instructions to initialize Log4j 2 in a web application.

15.1.1.3 Java EE Applications

A Java EE application will consist of one or more WAR files and possible some EJBs, typically all packaged in an EAR file. Usually, it is desirable to have a single configuration that applies to all the components in the EAR. The logging classes will generally be placed in a location shared across all the components and the configuration needs to also be shareable. Be sure to follow the instructions to initialize Log4j 2 in a web application.

15.1.1.4 "Shared" Web Applications and REST Service Containers

In this scenario there are multiple WAR files deployed into a single container. Each of the applications should use the same logging configuration and share the same logging implementation across each of the web applications. When writing to files and streams each of the applications should share them to avoid the issues that can occur when multiple components try to write to the same file(s) through different File objects, channels, etc.

15.1.1.5 OSGi Applications

An OSGi container physically separates each JAR into its own ClassLoader, thus enforcing modularity of JARs as well as providing standardized ways for JARs to share code based on version numbers. Suffice to say, the OSGi framework is beyond the scope of this manual. There are some differences when using Log4j in an OSGi container. By default, each JAR bundle is scanned for its own Log4j configuration file. Similar to the web application paradigm, every bundle has its own LoggerContext. As this may be undesirable when a global Log4j configuration is wanted, then the ContextSelector should be overridden with BasicContextSelector or JndiContextSelector.

15.1.2 Approaches
15.1.2.1 The Simple Approach

The simplest approach for separating logging within applications is to package each application with its own copy of Log4j and to use the BasicContextSelector. While this works for standalone applications and may work for web applications and possibly Java EE applications, it does not work at all in the last case. However, when this approach does work it should be used as it is ultimately the simplest and most straightforward way of implementing logging.

15.1.2.2 Using Context Selectors

There are a few patterns for achieving the desired state of logging separation using ContextSelectors:

1. Place the logging jars in the container's classpath and set the system property `log4j2.contextSelector` to `org.apache.logging.log4j.core.selector.BasicContextSelector`. This will create a single LoggerContext using a single configuration that will be shared across all applications.

2. Place the logging jars in the container's classpath and use the default `ClassLoaderContextSelector`. Follow the instructions to initialize Log4j 2 in a web application. Each application can be configured to share the same configuration used at the container or can be individually configured. If status logging is set to debug in the configuration there will be output from when logging is initialized in the container and then again in each web application.

3. Follow the instructions to initialize Log4j 2 in a web application and set the system property or servlet context parameter `log4j2.contextSelector` to `org.apache.logging.log4j.core.selector.JndiContextSelector`. This will cause the container to use JNDI to locate each web application's LoggerContext. Be sure to set the `isLog4jContextSelectorNamed` context parameter to true and also set the `log4jContextName` and `log4jConfiguration` context parameters.

The exact method for setting system properties depends on the container. For Tomcat, edit `$CATALINA_HOME/conf/catalina.properties`. Consult the documentation for other web containers.
16 Extending Log4j

16.1 Extending Log4j

Log4j 2 provides numerous ways that it can be manipulated and extended. This section includes an overview of the various ways that are directly supported by the Log4j 2 implementation.

16.1.1 LoggerContextFactory

The LoggerContextFactory binds the Log4j API to its implementation. The Log4j LogManager locates a LoggerContextFactory by using java.util.ServiceLoader to locate all instances of org.apache.logging.log4j.spi.Provider. Each implementation must provide a class that extends org.apache.logging.log4j.spi.Provider and should have a no-arg constructor that delegates to Provider's constructor passing the Priority, the API versions it is compatible with, and the class that implements org.apache.logging.log4j.spi.LoggerContextFactory. Log4j will compare the current API version and if it is compatible the implementation will be added to the list of providers. The API version in org.apache.logging.log4j.LogManager is only changed when a feature is added to the API that implementations need to be aware of. If more than one valid implementation is located the value for the Priority will be used to identify the factory with the highest priority. Finally, the class that implements org.apache.logging.log4j.spi.LoggerContextFactory will be instantiated and bound to the LogManager. In Log4j 2 this is provided by Log4jContextFactory.

Applications may change the LoggerContextFactory that will be used by

1. Create a binding to the logging implementation.
   b. Implement a class that extends org.apache.logging.spi.Provider with a no-arg constructor that calls super-class's constructor with the Priority, the API version(s), LoggerContextFactory class, and optionally, a ThreadContextMap implementation class.
   c. Create a META-INF/services/org.apache.logging.spi.Provider file that contains the name of the class that implements org.apache.logging.spi.Provider.
2. Setting the system property log4j2.loggerContextFactory to the name of the LoggerContextFactory class to use.
3. Setting the property "log4j2.loggerContextFactory" in a properties file named "log4j2.LogManager.properties" to the name of the LoggerContextFactory class to use. The properties file must be on the classpath.

16.1.2 ContextSelector

ContextSelectors are called by the Log4j LoggerContextFactory. They perform the actual work of locating or creating a LoggerContext, which is the anchor for Loggers and their configuration. ContextSelectors are free to implement any mechanism they desire to manage LoggerContexts. The default Log4jContextFactory checks for the presence of a System Property named "Log4jContextSelector". If found, the property is expected to contain the name of the Class that implements the ContextSelector to be used.

Log4j provides five ContextSelectors:

BasicContextSelector

Uses either a LoggerContext that has been stored in a ThreadLocal or a common LoggerContext.
ClassLoaderContextSelector

Associates LoggerContexts with the ClassLoader that created the caller of the getLogger call. This is the default ContextSelector.

JndiContextSelector

Locates the LoggerContext by querying JNDI.

AsyncLoggerContextSelector

Creates a LoggerContext that ensures that all loggers are AsyncLoggers.

BundleContextSelector

Associates LoggerContexts with the ClassLoader of the bundle that created the caller of the getLogger call. This is enabled by default in OSGi environments.

16.1.3 ConfigurationFactory

Modifying the way in which logging can be configured is usually one of the areas with the most interest. The primary method for doing that is by implementing or extending a ConfigurationFactory. Log4j provides two ways of adding new ConfigurationFactories. The first is by defining the system property named "log4j.configurationFactory" to the name of the class that should be searched first for a configuration. The second method is by defining the ConfigurationFactory as a Plugin.

All the ConfigurationFactories are then processed in order. Each factory is called on its getSupportedTypes method to determine the file extensions it supports. If a configuration file is located with one of the specified file extensions then control is passed to that ConfigurationFactory to load the configuration and create the Configuration object.

Most Configuration extend the BaseConfiguration class. This class expects that the subclass will process the configuration file and create a hierarchy of Node objects. Each Node is fairly simple in that it consists of the name of the node, the name/value pairs associated with the node, The PluginType of the node and a List of all of its child Nodes. BaseConfiguration will then be passed the Node tree and instantiate the configuration objects from that.
@Plugin(name = "XMLConfigurationFactory", category = "ConfigurationFactory")
@Order(5)
public class XMLConfigurationFactory extends ConfigurationFactory {

/**
 * Valid file extensions for XML files.
 */
public static final String[] SUFFIXES = new String[] {"*.xml", "*"};

/**
 * Returns the Configuration.
 * @param loggerContext The logger context.
 * @param source The InputSource.
 * @return The Configuration.
 */
@Override
public Configuration getConfiguration(final LoggerContext loggerContext, final ConfigurationSource source) {
    return new XmlConfiguration(loggerContext, source);
}

/**
 * Returns the file suffixes for XML files.
 * @return An array of File extensions.
 */
public String[] getSupportedTypes() {
    return SUFFIXES;
}
}

16.1.4 LoggerConfig

LoggerConfig objects are where Loggers created by applications tie into the configuration. The Log4j implementation requires that all LoggerConfigs be based on the LoggerConfig class, so applications wishing to make changes must do so by extending the LoggerConfig class. To declare the new LoggerConfig, declare it as a Plugin of type "Core" and providing the name that applications should specify as the element name in the configuration. The LoggerConfig should also define a PluginFactory that will create an instance of the LoggerConfig.

The following example shows how the root LoggerConfig simply extends a generic LoggerConfig.

@Plugin(name = "root", category = "Core", printObject = true)
public static class RootLogger extends LoggerConfig {

    @PluginFactory
    public static LoggerConfig createLogger(@PluginAttribute(value = "additivity", defaultBooleanValue = true) boolean additivity,
                                            @PluginAttribute(value = "level", defaultStringValue = "ERROR") Level level,
                                            @PluginElement("AppenderRef") AppenderRef[] refs,
                                            @PluginElement("Filters") Filter filter) {
        List<AppenderRef> appenderRefs = Arrays.asList(refs);
        return new LoggerConfig(LogManager.ROOT_LOGGER_NAME, appenderRefs, filter, level, additivity);
    }
}

©2019, The Apache Software Foundation • ALL RIGHTS RESERVED.
16.1.5 LogEventFactory

A LogEventFactory is used to generate LogEvents. Applications may replace the standard LogEventFactory by setting the value of the system property Log4jLogEventFactory to the name of the custom LogEventFactory class.

Note: When log4j is configured to have all loggers asynchronous, log events are pre-allocated in a ring buffer and the LogEventFactory is not used.

16.1.6 MessageFactory

A MessageFactory is used to generate Message objects. Applications may replace the standard ParameterizedMessageFactory (or ReusableMessageFactory in garbage-free mode) by setting the value of the system property log4j2.messageFactory to the name of the custom MessageFactory class.

Flow messages for the Logger.entry() and Logger.exit() methods have a separate FlowMessageFactory. Applications may replace the DefaultFlowMessageFactory by setting the value of the system property log4j2.flowMessageFactory to the name of the custom FlowMessageFactory class.

16.1.7 Lookups

Lookups are the means in which parameter substitution is performed. During Configuration initialization an "Interpolator" is created that locates all the Lookups and registers them for use when a variable needs to be resolved. The interpolator matches the "prefix" portion of the variable name to a registered Lookup and passes control to it to resolve the variable.

A Lookup must be declared using a Plugin annotation with a type of "Lookup". The name specified on the Plugin annotation will be used to match the prefix. Unlike other Plugins, Lookups do not use a PluginFactory. Instead, they are required to provide a constructor that accepts no arguments. The example below shows a Lookup that will return the value of a System Property.

The provided Lookups are documented here: Lookups

```java
@Plugin(name = "sys", category = "Lookup")
public class SystemPropertiesLookup implements StrLookup {

    /**
     * Lookup the value for the key.
     * @param key  the key to be looked up, may be null
     * @return The value for the key.
     */
    public String lookup(String key) {
        return System.getProperty(key);
    }

    /**
     * Lookup the value for the key using the data in the LogEvent.
     * @param event The current LogEvent.
     * @param key  the key to be looked up, may be null
     * @return The value associated with the key.
     */
    public String lookup(LogEvent event, String key) {
        return System.getProperty(key);
    }
}
```
16.1.8 Filters

As might be expected, Filters are used to reject or accept log events as they pass through the logging system. A Filter is declared using a Plugin annotation of type "Core" and an elementType of "filter". The name attribute on the Plugin annotation is used to specify the name of the element users should use to enable the Filter. Specifying the printObject attribute with a value of "true" indicates that a call to toString will format the arguments to the filter as the configuration is being processed. The Filter must also specify a PluginFactory method that will be called to create the Filter.

The example below shows a Filter used to reject LogEvents based upon their logging level. Notice the typical pattern where all the filter methods resolve to a single filter method.
public final class ThresholdFilter extends AbstractFilter {
    private final Level level;

    private ThresholdFilter(Level level, Result onMatch, Result onMismatch) {
        super(onMatch, onMismatch);
        this.level = level;
    }

    public Result filter(Logger logger, Level level, Marker marker, String msg, Object[] params) {
        return filter(level);
    }

    public Result filter(Logger logger, Level level, Marker marker, Object msg, Throwable t) {
        return filter(level);
    }

    public Result filter(Logger logger, Level level, Marker marker, Message msg, Throwable t) {
        return filter(level);
    }

    @Override
    public Result filter(LogEvent event) {
        return filter(event.getLevel());
    }

    private Result filter(Level level) {
        return level.isAtLeastAsSpecificAs(this.level) ? onMatch : onMismatch;
    }

    @Override
    public String toString() {
        return level.toString();
    }

    /**
     * Create a ThresholdFilter.
     * @param loggerLevel The log Level.
     * @param match The action to take on a match.
     * @param mismatch The action to take on a mismatch.
     * @return The created ThresholdFilter.
     */
    @PluginFactory
    public static ThresholdFilter createFilter(@PluginAttribute(value = "level", defaultStringValue = "ERROR"
            @PluginAttribute(value = "onMatch", defaultStringValue = "NEUTRAL"
            @PluginAttribute(value = "onMismatch", defaultStringValue = "DENY"
16.1.9 Appenders

Appenders are passed an event, (usually) invoke a Layout to format the event, and then "publish" the event in whatever manner is desired. Appenders are declared as Plugins with a type of "Core" and an elementType of "appender". The name attribute on the Plugin annotation specifies the name of the element users must provide in their configuration to use the Appender. Appenders should specify printObject as "true" if the toString method renders the values of the attributes passed to the Appender.

Appenders must also declare a PluginFactory method that will create the appender. The example below shows an Appender named "Stub" that can be used as an initial template.

Most Appenders use Managers. A manager actually "owns" the resources, such as an OutputStream or socket. When a reconfiguration occurs a new Appender will be created. However, if nothing significant in the previous Manager has changed, the new Appender will simply reference it instead of creating a new one. This insures that events are not lost while a reconfiguration is taking place without requiring that logging pause while the reconfiguration takes place.

```
@Plugin(name = "Stub", category = "Core", elementType = "appender", printObject = true)
public final class StubAppender extends AbstractOutputStreamAppender<StubManager> {

    private StubAppender(String name,
                        Layout<? extends Serializable> layout,
                        Filter filter,
                        boolean ignoreExceptions,
                        StubManager manager) {
        super(name, layout, filter, ignoreExceptions, true, manager);
    }

    @PluginFactory
    public static StubAppender createAppender(@PluginAttribute("name") String name,
                                              @PluginAttribute("ignoreExceptions") boolean ignoreExceptions,
                                              @PluginElement("Layout") Layout layout,
                                              @PluginElement("Filters") Filter filter) {

        if (name == null) {
            LOGGER.error("No name provided for StubAppender");
            return null;
        }

        StubManager manager = StubManager.getStubManager(name);
        if (manager == null) {
            return null;
        }

        if (layout == null) {
            layout = PatternLayout.createDefaultLayout();
        }

        return new StubAppender(name, layout, filter, ignoreExceptions, manager);
    }
}
```

16.1.10 Layouts

Layouts perform the formatting of events into the printable text that is written by Appenders to some destination. All Layouts must implement the Layout interface. Layouts that format the event into a
String should extend AbstractStringLayout, which will take care of converting the String into the required byte array.

Every Layout must declare itself as a plugin using the Plugin annotation. The type must be "Core", and the elementType must be "layout". printObject should be set to true if the plugin's toString method will provide a representation of the object and its parameters. The name of the plugin must match the value users should use to specify it as an element in their Appender configuration. The plugin also must provide a static method annotated as a PluginFactory and with each of the methods parameters annotated with PluginAttr or PluginElement as appropriate.

@Plugin(name = "SampleLayout", category = "Core", elementType = "layout", printObject = true)
public class SampleLayout extends AbstractStringLayout {

    protected SampleLayout(boolean locationInfo, boolean properties, boolean complete,
                          Charset charset) {
    }

    @PluginFactory
    public static SampleLayout createLayout(@PluginAttribute("locationInfo") boolean locationInfo,
                                             @PluginAttribute("properties") boolean properties,
                                             @PluginAttribute("complete") boolean complete,
                                             @PluginAttribute(value = "charset", defaultStringValue = "UTF-8") Charset charset) {
        return new SampleLayout(locationInfo, properties, complete, charset);
    }
}

16.1.11 PatternConverters

PatternConverters are used by the PatternLayout to format the log event into a printable String. Each Converter is responsible for a single kind of manipulation, however Converters are free to format the event in complex ways. For example, there are several converters that manipulate Throwables and format them in various ways.

A PatternConverter must first declare itself as a Plugin using the standard Plugin annotation but must specify value of "Converter" on the type attribute. Furthermore, the Converter must also specify the ConverterKeys attribute to define the tokens that can be specified in the pattern (preceded by a '%' character) to identify the Converter.

Unlike most other Plugins, Converters do not use a PluginFactory. Instead, each Converter is required to provide a static newInstance method that accepts an array of Strings as the only parameter. The String array are the values that are specified within the curly braces that can follow the converter key.

The following shows the skeleton of a Converter plugin.
@Plugin(name = "query", category = "Converter")
@ConverterKeys({"q", "query"})
public final class QueryConverter extends LogEventPatternConverter {

    public QueryConverter(String[] options) {
    }

    public static QueryConverter newInstance(final String[] options) {
        return new QueryConverter(options);
    }
}

### 16.1.12 Plugin Builders

Some plugins take a lot of optional configuration options. When a plugin takes many options, it is more maintainable to use a builder class rather than a factory method (see Item 2: Consider a builder when faced with many constructor parameters in Effective Java by Joshua Bloch). There are some other advantages to using an annotated builder class over an annotated factory method:

- Attribute names don’t need to be specified if they match the field name.
- Default values can be specified in code rather than through an annotation (also allowing a runtime-calculated default value which isn’t allowed in annotations).
- Adding new optional parameters doesn’t require existing programmatic configuration to be refactored.
- Easier to write unit tests using builders rather than factory methods with optional parameters.
- Default values are specified via code rather than relying on reflection and injection, so they work programmatically as well as in a configuration file.

Here is an example of a plugin factory from `ListAppender`:

```java
@PluginFactory
public static ListAppender createAppender(
    @PluginAttribute("name") @Required(message = "No name provided for ListAppender") final String name,
    @PluginAttribute("entryPerNewLine") final boolean newLine,
    @PluginAttribute("raw") final boolean raw,
    @PluginElement("Layout") final Layout<? extends Serializable> layout,
    @PluginElement("Filter") final Filter filter) {
    return new ListAppender(name, filter, layout, newLine, raw);
}
```

Here is that same factory using a builder pattern instead:
@PluginBuilderFactory
public static Builder newBuilder() {
    return new Builder();
}

public static class Builder implements org.apache.logging.log4j.core.util.Builder<ListAppender> {
    @PluginBuilderAttribute
    @Required(message = "No name provided for ListAppender")
    private String name;

    @PluginBuilderAttribute
    private boolean entryPerNewLine;

    @PluginBuilderAttribute
    private boolean raw;

    @PluginElement("Layout")
    private Layout<? extends Serializable> layout;

    @PluginElement("Filter")
    private Filter filter;

    public Builder setName(final String name) {
        this.name = name;
        return this;
    }

    public Builder setEntryPerNewLine(final boolean entryPerNewLine) {
        this.entryPerNewLine = entryPerNewLine;
        return this;
    }

    public Builder setRaw(final boolean raw) {
        this.raw = raw;
        return this;
    }

    public Builder setLayout(final Layout<? extends Serializable> layout) {
        this.layout = layout;
        return this;
    }

    public Builder setFilter(final Filter filter) {
        this.filter = filter;
        return this;
    }

    @Override
    public ListAppender build() {
        return new ListAppender(name, filter, layout, entryPerNewLine, raw);
    }
}
The only difference in annotations is using @PluginBuilderAttribute instead of @PluginAttribute so that default values and reflection can be used instead of specifying them in the annotation. Either annotation can be used in a builder, but the former is better suited for field injection while the latter is better suited for parameter injection. Otherwise, the same annotations (@PluginConfiguration, @PluginElement, @PluginNode, and @PluginValue) are all supported on fields. Note that a factory method is still required to supply a builder, and this factory method should be annotated with @PluginBuilderFactory.

When plugins are being constructed after a configuration has been parsed, a plugin builder will be used if available, otherwise a plugin factory method will be used as a fallback. If a plugin contains neither factory, then it cannot be used from a configuration file (it can still be used programmatically of course).

Here is an example of using a plugin factory versus a plugin builder programmatically:

```java
ListAppender list1 = ListAppender.createAppender("List1", true, false, null, null);
ListAppender list2 = ListAppender.newBuilder().setName("List1").setEntryPerNewLine(true).build();
```

16.1.13 Custom ContextDataInjector

The ContextDataInjector (introduced in Log4j 2.7) is responsible for populating the LogEvent's context data with key-value pairs or replacing it completely. The default implementation is ThreadContextDataInjector, which obtains context attributes from the ThreadContext.

Applications may replace the default ContextDataInjector by setting the value of the system property log4j2.contextDataInjector to the name of the custom ContextDataInjector class.

Implementors should be aware there are some subtleties related to thread-safety and implementing a context data injector in a garbage-free manner. See the ContextDataInjector javadoc for detail.

16.1.14 Custom ThreadContextMap implementations

A garbage-free StringMap-based context map can be installed by setting system property log4j2.garbageFreeThreadContextMap to true. (Log4j must be enabled to use ThreadLocals.)

Any custom ThreadContextMap implementation can be installed by setting system property log4j2.threadContextMap to the fully qualified class name of the class implementing the ThreadContextMap interface. By also implementing the ReadOnlyThreadContextMap interface, your custom ThreadContextMap implementation will be accessible to applications via the ThreadContext::getThreadContextMap method.

16.1.15 Custom Plugins

See the Plugins section of the manual.
17 Programmatic Log4j Configuration

17.1 Programmatic Configuration
Log4j 2 provides a few ways for applications to create their own programmatic configuration:

- Specify a custom ConfigurationFactory to start Log4j with a programmatic configuration
- Use the Configurator to replace the configuration after Log4j started
- Initialize Log4j with a combination of a configuration file and programmatic configuration
- Modify the current Configuration after initialization

17.1.1 The ConfigurationBuilder API
Starting with release 2.4, Log4j provides a ConfigurationBuilder and a set of component builders that allow a Configuration to be created fairly easily. Actual configuration objects like LoggerConfig or Appender can be unwieldy; they require a lot of knowledge about Log4j internals which makes them difficult to work with if all you want is to create a Configuration.

The new ConfigurationBuilder API (in the org.apache.logging.log4j.core.config.builder.api package) allows users to create Configurations in code by constructing component definitions. There is no need to work directly with actual configuration objects. Component definitions are added to the ConfigurationBuilder, and once all the definitions have been collected all the actual configuration objects (like Loggers and Appenders) are constructed.

ConfigurationBuilder has convenience methods for the base components that can be configured such as Loggers, Appenders, Filter, Properties, etc. However, Log4j 2's plugin mechanism means that users can create any number of custom components. As a trade-off, the ConfigurationBuilder API provides only a limited number of "strongly typed" convenience methods like newLogger(), newLayout() etc. The generic builder.newComponent() method can be used if no convenience method exists for the component you want to configure.

For example, the builder does not know what sub-components can be configured on specific components such as the RollingFileAppender vs. the RoutingAppender. To specify a triggering policy on a RollingFileAppender you would use builder.newComponent().

Examples of using the ConfigurationBuilder API are in the sections that follow.

17.1.2 Understanding ConfigurationFactory
During initialization, Log4j 2 will search for available ConfigurationFactories and then select the one to use. The selected ConfigurationFactory creates the Configuration that Log4j will use. Here is how Log4j finds the available ConfigurationFactories:

1. A system property named log4j2.configurationFactory can be set with the name of the ConfigurationFactory to be used.
2. ConfigurationFactory.setConfigurationFactory(ConfigurationFactory) can be called with the instance of the ConfigurationFactory to be used. This must be called before any other calls to Log4j.
3. A ConfigurationFactory implementation can be added to the classpath and configured as a plugin in the "ConfigurationFactory" category. The Order annotation can be used to specify the relative priority when multiple applicable ConfigurationFactories are found.
ConfigurationFactories have the concept of "supported types", which basically maps to the file extension of the configuration file that the ConfigurationFactory can handle. If a configuration file location is specified, ConfigurationFactories whose supported type does not include ".*" or the matching file extension will not be used.

17.1.3 Initialize Log4j Using ConfigurationBuilder with a Custom ConfigurationFactory

One way to programmatically configure Log4j 2 is to create a custom ConfigurationFactory that uses the ConfigurationBuilder to create a Configuration. The below example overrides the getConfiguration() method to return a Configuration created by the ConfigurationBuilder. This will cause the Configuration to automatically be hooked into Log4j when the LoggerContext is created. In the example below, because it specifies a supported type of ".*" it will override any configuration files provided.
@Plugin(name = "CustomConfigurationFactory", category = ConfigurationFactory.CATEGORY)
@Order(50)
public class CustomConfigurationFactory extends ConfigurationFactory {

    static Configuration createConfiguration(final String name, ConfigurationBuilder<BuiltConfiguration> builder) {
        builder.setConfigurationName(name);
        builder.setStatusLevel(Level.ERROR);
            addAttribute("level", Level.DEBUG));
        AppendableComponentBuilder appenderBuilder = builder.newAppender("Stdout", "CONSOLE").
            addAttribute("target", ConsoleAppender.Target.SYSTEM_OUT);
        appenderBuilder.add(builder.newLayout("PatternLayout").
            addAttribute("pattern", "%d [%t] %-5level: %msg%n%throwable").
        appenderBuilder.add(builder.newFilter("MarkerFilter", Filter.Result.DENY,
            Filter.Result.NEUTRAL).addAttribute("marker", "FLOW").
        builder.add(appenderBuilder);
            add(builder.newAppenderRef("Stdout")).
            addAttribute("additivity", false));
        builder.add(builder.newRootLogger(Level.ERROR).add(builder.newAppenderRef("Stdout")));
        return builder.build();
    }

    @Override
    public Configuration getConfiguration(final LoggerContext loggerContext, final ConfigurationSource source) {
        return getConfiguration(loggerContext, source.toString(), null);
    }

    @Override
    public Configuration getConfiguration(final LoggerContext loggerContext, final String name, final URI configLocation) {
        ConfigurationBuilder<BuiltConfiguration> builder = newConfigurationBuilder();
        return createConfiguration(name, builder);
    }

    @Override
    protected String[] getSupportedTypes() {
        return new String[] {"*"};
    }
}

As of version 2.7, the ConfigurationFactory.getConfiguration() methods take an additional LoggerContext parameter.

17.1.4 Reconfigure Log4j Using ConfigurationBuilder with the Configurator

An alternative to a custom ConfigurationFactory is to configure with the Configurator. Once a Configuration object has been constructed, it can be passed to one of the Configurator.initialize methods to set up the Log4j configuration.

Using the Configurator in this manner allows the application control over when Log4j is initialized. However, should any logging be attempted before Configurator.initialize() is called then the default configuration will be used for those log events.
ConfigurationBuilder<BuiltConfiguration> builder = ConfigurationBuilderFactory.newConfigurationBuilder();
builder.setStatusLevel(Level.ERROR);
builder.setConfigurationName("BuilderTest");
  .addAttribute("level", Level.DEBUG));
AppenderComponentBuilder appenderBuilder = builder.newAppender("Stdout", "CONSOLE").addAttribute("target", ConsoleAppender.Target.SYSTEM_OUT);
appenderBuilder.add(builder.newLayout("PatternLayout")
  .addAttribute("pattern", "%d [%t] %-5level: %msg%n%throwable");
  .addAttribute("marker", "FLOW");
builder.add(appenderBuilder);
builder.add(builder.newLogger("org.apache.logging.log4j", Level.DEBUG)
  .add(builder.newAppenderRef("Stdout")).addAttribute("additivity", false));
ctx = Configurator.initialize(builder.build());

This example shows how to create a configuration that includes a RollingFileAppender.

ConfigurationBuilder<BuiltConfiguration> builder = ConfigurationBuilderFactory.newConfigurationBuilder();

builder.setConfigurationName("RollingBuilder");
builder.setConfigurationName("RollingBuilder");
// create a console appender
AppenderComponentBuilder appenderBuilder = builder.newAppender("Stdout", "CONSOLE").addAttribute("target", ConsoleAppender.Target.SYSTEM_OUT);
appenderBuilder.add(builder.newLayout("PatternLayout")
  .addAttribute("pattern", "%d [%t] %-5level: %msg%n%throwable");
builder.add(appenderBuilder);
// create a rolling file appender
LayoutComponentBuilder layoutBuilder = builder.newLayout("PatternLayout")
  .addAttribute("pattern", "%d [%t] %-5level: %msg%n");
ComponentBuilder triggeringPolicy = builder.newComponent("Policies")
  .addComponent(builder.newComponent("CronTriggeringPolicy").addAttribute("schedule", "0 0 0 * * ?")
  .addComponent(builder.newComponent("SizeBasedTriggeringPolicy").addAttribute("size", "100M");
appenderBuilder = builder.newAppender("rolling", "RollingFile")
  .addAttribute("fileName", "target/rolling.log")
  .addAttribute("filePattern", "target/archive/rolling-%d{MM-dd-yy}.log.gz")
  .add(layoutBuilder)
  .addComponent(triggeringPolicy);
builder.add(appenderBuilder);

// create the new logger
builder.add(builder.newLogger("TestLogger", Level.DEBUG)
  .add(builder.newAppenderRef("rolling");
  .addAttribute("additivity", false));

builder.add(builder.newRootLogger(Level.DEBUG)
  .add(builder.newAppenderRef("rolling");
LoggerContext ctx = Configurator.initialize(builder.build());
17.1.5 Initialize Log4j by Combining Configuration File with Programmatic Configuration

Sometimes you want to configure with a configuration file but do some additional programmatic configuration. A possible use case might be that you want to allow for a flexible configuration using XML but at the same time make sure there are a few configuration elements that are always present that can't be removed.

The easiest way to achieve this is to extend one of the standard Configuration classes (XMLConfiguration, JSONConfiguration) and then create a new ConfigurationFactory for the extended class. After the standard configuration completes the custom configuration can be added to it.

The example below shows how to extend XMLConfiguration to manually add an Appender and a LoggerConfig to the configuration.
@Plugin(name = "MyXMLConfigurationFactory", category = "ConfigurationFactory")
@Order(10)
public class MyXMLConfigurationFactory extends ConfigurationFactory {

    /**
     * Valid file extensions for XML files.
     */
    public static final String[] SUFFIXES = new String[] {"*.xml", "**"};

    /**
     * Return the Configuration.
     * @param source The InputSource.
     * @return The Configuration.
     */
    public Configuration getConfiguration(InputSource source) {
        return new MyXMLConfiguration(source, configFile);
    }

    /**
     * Returns the file suffixes for XML files.
     * @return An array of File extensions.
     */
    public String[] getSupportedTypes() {
        return SUFFIXES;
    }
}

public class MyXMLConfiguration extends XMLConfiguration {
    public MyXMLConfiguration(final ConfigurationFactory.ConfigurationSource configSource) {
        super(configSource);
    }

    @Override
    protected void doConfigure() {
        super.doConfigure();
        final LoggerContext ctx = (LoggerContext) LogManager.getContext(false);
        final Layout layout = PatternLayout.createLayout(PatternLayout.SIMPLE_CONVERSION_PATTERN, config, null, null, null, null);
        final Appender appender = FileAppender.createAppender("target/test.log", "false", "false", "File", "true", "false", "false", "4000", layout, null, "false", null, config);
        appender.start();
        addAppender(appender);
        LoggerConfig loggerConfig = LoggerConfig.createLogger("false", "info", "org.apache.logging.log4j", "true", refs, null, config, null);
        loggerConfig.addAppender(appender, null, null);
        addLogger("org.apache.logging.log4j", loggerConfig);
    }
}
17.1.6 Programmatically Modifying the Current Configuration after Initialization

Applications sometimes have the need to customize logging separate from the actual configuration. Log4j allows this although it suffers from a few limitations:

1. If the configuration file is changed the configuration will be reloaded and the manual changes will be lost.
2. Modification to the running configuration requires that all the methods being called (addAppender and addLogger) be synchronized.

As such, the recommended approach for customizing a configuration is to extend one of the standard Configuration classes, override the setup method to first do super.setup() and then add the custom Appenders, Filters and LoggerConfigs to the configuration before it is registered for use.

The following example adds an Appender and a new LoggerConfig using that Appender to the current configuration.

```java
final LoggerContext ctx = (LoggerContext) LogManager.getContext(false);
final Configuration config = ctx.getConfiguration();
Layout layout = PatternLayout.createLayout(PatternLayout.SIMPLE_CONVERSION_PATTERN, config, null, null, null, null);
Appender appender = FileAppender.createAppender("target/test.log", "false", "false", "File", "true", "false", "false", "4000", layout, null, "false", null, config);
appender.start();
config.addAppender(appender);
AppenderRef[] refs = new AppenderRef[] {ref};
LoggerConfig loggerConfig = LoggerConfig.createLogger("false", "info", "org.apache.logging.log4j", "true", refs, null, config, null );
loggerConfig.addAppender(appender, null, null);
config.addLogger("org.apache.logging.log4j", loggerConfig);
ctx.updateLoggers();
}
```

17.1.7 Appending Log Events to Writers and OutputStreams Programatically

Log4j 2.5 provides facilities to append log events to Writers and OutputStreams. For example, this provides simple integration for JDBC Driver implementors that use Log4j internally and still want to support the JDBC APIs `CommonDataSource.setLogWriter(PrintWriter)`, `java.sql.DriverManager.setLogWriter(PrintWriter)`, and `java.sql.DriverManager.setLogStream(PrintStream)`.

Given any Writer, like a PrintWriter, you tell Log4j to append events to that writer by creating a WriterAppender and updating the Log4j configuration:
```java
void addAppender(final Writer writer, final String writerName) {
    final LoggerContext context = LoggerContext.getContext(false);
    final Configuration config = context.getConfiguration();
    final PatternLayout layout = PatternLayout.createDefaultLayout(config);
    final Appender appender = WriterAppender.createAppender(layout, null, writer, writerName, false, true);
    appender.start();
    config.addAppender(appender);
    updateLoggers(appender, config);
}

private void updateLoggers(final Appender appender, final Configuration config) {
    final Level level = null;
    final Filter filter = null;
    for (final LoggerConfig loggerConfig : config.getLoggers().values()) {
        loggerConfig.addAppender(appender, level, filter);
    }
    config.getRootLogger().addAppender(appender, level, filter);
}

You can achieve the same effect with an OutputStream, like a PrintStream:

```java
void addAppender(final OutputStream outputStream, final String outputStreamName) {
    final LoggerContext context = LoggerContext.getContext(false);
    final Configuration config = context.getConfiguration();
    final PatternLayout layout = PatternLayout.createDefaultLayout(config);
    final Appender appender = OutputStreamAppender.createAppender(layout, null, outputStream, outputStreamName, false, true);
    appender.start();
    config.addAppender(appender);
    updateLoggers(appender, config);
}
```

The difference is the use of OutputStreamAppender instead of WriterAppender.
18 Custom Log Levels

18.1 Custom Log Levels

18.1.1 Defining Custom Log Levels in Code

Log4J 2 supports custom log levels. Custom log levels can be defined in code or in configuration. To define a custom log level in code, use the `Level.forName()` method. This method creates a new level for the specified name. After a log level is defined you can log messages at this level by calling the `Logger.log()` method and passing the custom log level:

```java
// This creates the "VERBOSE" level if it does not exist yet.
final Level VERBOSE = Level.forName("VERBOSE", 550);

final Logger logger = LogManager.getLogger();
logger.log(VERBOSE, "a verbose message"); // use the custom VERBOSE level

// Create and use a new custom level "DIAG".
logger.log(Level.forName("DIAG", 350), "a diagnostic message");

// Use (don't create) the "DIAG" custom level.
// Only do this *after* the custom level is created!
logger.log(Level.getLevel("DIAG"), "another diagnostic message");

// Using an undefined level results in an error: Level.getLevel() returns null,
// and logger.log(null, "message") throws an exception.
logger.log(Level.getLevel("FORGOT_TO_DEFINE"), "some message"); // throws exception!
```

When defining a custom log level, the `intLevel` parameter (550 and 350 in the example above) determines where the custom level exists in relation to the standard levels built-in to Log4J 2. For reference, the table below shows the `intLevel` of the built-in log levels.

<table>
<thead>
<tr>
<th>Standard Level</th>
<th>intLevel</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>0</td>
</tr>
<tr>
<td>FATAL</td>
<td>100</td>
</tr>
<tr>
<td>ERROR</td>
<td>200</td>
</tr>
<tr>
<td>WARN</td>
<td>300</td>
</tr>
<tr>
<td>INFO</td>
<td>400</td>
</tr>
<tr>
<td>DEBUG</td>
<td>500</td>
</tr>
<tr>
<td>TRACE</td>
<td>600</td>
</tr>
<tr>
<td>ALL</td>
<td>Integer.MAX_VALUE</td>
</tr>
</tbody>
</table>

Standard log levels built-in to Log4J
Custom log levels can also be defined in configuration. This is convenient for using a custom level in a logger filter or an appender filter. Similar to defining log levels in code, a custom level must be defined first, before it can be used. If a logger or appender is configured with an undefined level, that logger or appender will be invalid and will not process any log events.

The `CustomLevel` configuration element creates a custom level. Internally it calls the same `Level.forName()` method discussed above.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the custom level. Note that level names are case sensitive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The convention is to use all uppercase names.</td>
</tr>
<tr>
<td>intLevel</td>
<td>integer</td>
<td>Determines where the custom level exists in relation to the standard levels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>built-in to Log4J 2 (see the table above).</td>
</tr>
</tbody>
</table>

**CustomLevel Parameters**

The following example shows a configuration that defines some custom log levels and uses a custom log level to filter log events sent to the console.

```xml
<Configuration status="WARN">
  <!-- Define custom levels before using them for filtering below. -->
  <CustomLevels>
    <CustomLevel name="DIAG" intLevel="350" />
    <CustomLevel name="NOTICE" intLevel="450" />
    <CustomLevel name="VERBOSE" intLevel="550" />
  </CustomLevels>

  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-7level %logger{36} - %msg%n"/>
    </Console>
    <File name="MyFile" fileName="logs/app.log">
      <PatternLayout pattern="%d %-7level %logger{36} - %msg%n"/>
    </File>
  </Appenders>

  <Loggers>
    <Root level="trace">
      <!-- Only events at DIAG level or more specific are sent to the console. -->
      <AppenderRef ref="Console" level="diag" />
      <AppenderRef ref="MyFile" level="trace" />
    </Root>
  </Loggers>
</Configuration>
```
18.1.3 Convenience Methods for the Built-in Log Levels

The built-in log levels have a set of convenience methods on the Logger interface that makes them easier to use. For example, the Logger interface has 24 `debug()` methods that support the DEBUG level:

```java
// convenience methods for the built-in DEBUG level
debug(Marker, Message)
debug(Marker, Message, Throwable)
debug(Marker, Object)
debug(Marker, Object, Throwable)
debug(Marker, String)
debug(Marker, String, Object...)
debug(Marker, String, Throwable)
debug(Message)
debug(Message, Throwable)
debug(Object)
debug(Object, Throwable)
debug(String)
debug(String, Object...)
debug(String, Throwable)
```

// lambda support methods added in 2.4
```
debug(Marker, MessageSupplier)
debug(Marker, MessageSupplier, Throwable)
debug(Marker, String, Supplier<?>...)
debug(Marker, Supplier<?>)
debug(Marker, Supplier<?>, Throwable)
debug(MessageSupplier)
debug(MessageSupplier, Throwable)
debug(String, Supplier<?>...)
debug(Supplier<?>)
debug(Supplier<?>, Throwable)
```

Similar methods exist for the other built-in levels. Custom levels, in contrast, need to pass in the log level as an extra parameter.

```java
// need to pass the custom level as a parameter
logger.log(VERBOSE, "a verbose message");
logger.log(Level.forName("DIAG", 350), "another message");
```

It would be nice to have the same ease of use with custom levels, so that after declaring the custom VERBOSE/DIAG levels, we could use code like this:

```java
// nice to have: descriptive methods and no need to pass the level as a parameter
logger.verbose("a verbose message");
logger.diag("another message");
logger.diag("java 8 lambda expression: ", () -> someMethod());
```

The standard Logger interface cannot provide convenience methods for custom levels, but the next few sections introduce a code generation tool to create loggers that aim to make custom levels as easy to use as built-in levels.
18.1.4 Adding or Replacing Log Levels

We assume that most users want to add custom level methods to the Logger interface, in addition to the existing trace(), debug(), info(), ... methods for the built-in log levels.

There is another use case, Domain Specific Language loggers, where we want to replace the existing trace(), debug(), info(), ... methods with all-custom methods.

For example, for medical devices we could have only critical(), warning(), and advisory() methods. Another example could be a game that has only defcon1(), defcon2(), and defcon3() levels.

If it were possible to hide existing log levels, users could customize the Logger interface to match their requirements. Some people may not want to have a FATAL or a TRACE level, for example. They would like to be able to create a custom Logger that only has debug(), info(), warn() and error() methods.

18.1.5 Generating Source Code for a Custom Logger Wrapper

Common Log4J usage is to get an instance of the Logger interface from the LogManager and call the methods on this interface. However, the custom log Levels are not known in advance, so Log4J cannot provide an interface with convenience methods for these custom log Levels.

To solve this, Log4J ships with a tool that generates source code for a Logger wrapper. The generated wrapper class has convenience methods for each custom log level, making custom levels just as easy to use as the built-in levels.

There are two flavors of wrappers: ones that extend the Logger API (adding methods to the built-in levels) and ones that customize the Logger API (replacing the built-in methods).

When generating the source code for a wrapper class, you need to specify:

- the fully qualified name of the class to generate
- the list of custom levels to support and their intLevel relative strength
- whether to extend Logger (and keep the existing built-in methods) or have only methods for the custom log levels

You would then include the generated source code in the project where you want to use custom log levels.

18.1.6 Example Usage of a Generated Logger Wrapper

Here is an example of how one would use a generated logger wrapper with custom levels DIAG, NOTICE and VERBOSE:
// ExtLogger is a generated logger wrapper
import com.mycompany.myproject.ExtLogger;

public class MyService {
    // instead of Logger logger = LogManager.getLogger(MyService.class):
    private static final ExtLogger logger = ExtLogger.create(MyService.class);

    public void demoExtendedLogger() {
        // ...
        logger.trace("the built-in TRACE level");
        logger.verbose("a custom level: a VERBOSE message");
        logger.debug("the built-in DEBUG level");
        logger.notice("a custom level: a NOTICE message");
        logger.info("the built-in INFO level");
        logger.diag("a custom level: a DIAG message");
        logger.warn("the built-in WARN level");
        logger.error("the built-in ERROR level");
        logger.fatal("the built-in FATAL level");
        logger.notice("java 8 lambda expression only executed if NOTICE is enabled: {}", () -> someMethod());
        // ...
    }
}

18.1.7 Generating Extended Loggers

Use the following command to generate a logger wrapper that adds methods to the built-in ones:

```bash
```

This will generate source code for a logger wrapper that has the convenience methods for the built-in levels as well as the specified custom levels. The tool prints the generated source code to the console. By appending " > filename" the output can be redirected to a file.

NOTE: Prior to log4j-2.9, this tool was an inner class Generate$ExtendedLogger.
Under the bash shell on Unix/Mac/Linux the dollar character $ needs to be escaped, so the class name should be between single quotes 'org.apache.logging.log4j.core.tools.Generate$ExtendedLogger'.

18.1.8 Generating Custom Loggers

Use the following command to generate a logger wrapper that hides the built-in levels and has only custom levels:

```bash
```

This will generate source code for a logger wrapper that only has convenience methods for the specified custom levels, not for the built-in levels. The tool prints the generated source code to the console. By appending " > filename" the output can be redirected to a file.

NOTE: Prior to log4j-2.9, this tool was an inner class Generate$ExtendedLogger.
Under the bash shell on Unix/Mac/Linux the dollar character $ needs to be escaped, so the class name should be between single quotes 'org.apache.logging.log4j.core.tools.Generate$CustomLogger'.