Apache Log4j 2

v. 2.3.2

User's Guide

The Apache Software Foundation **2021-12-29**

Table of Contents

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. 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 lock-free 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.

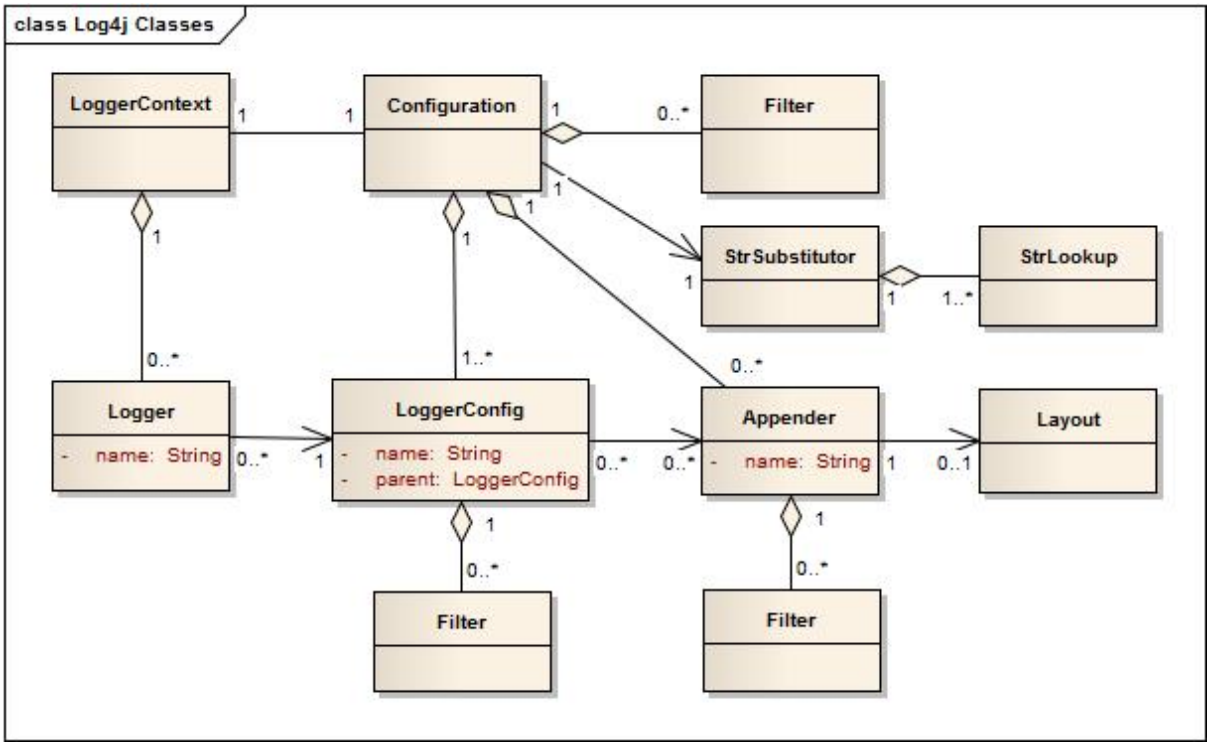
3. Log4j 2 uses a Plugin system that makes it extremely easy to extend the framework by adding new Appenders, Filters, Layouts, Lookups, and Pattern Converters without requiring any changes to Log4j.
4. Due to the Plugin system configuration is simpler. Entries in the configuration do not require a class name to be specified.
5. Support for custom log levels. Custom log levels can be defined in code or in configuration.
6. 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.
7. 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.
8. 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.
9. 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.
10. The Syslog Appender supports both TCP and UDP as well as support for the BSD syslog and the RFC 5424 formats.
11. 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.
12. 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.



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:

```
Logger logger = LogManager.getLogger(LogManager.ROOT_LOGGER_NAME);
```

Alternatively, and more simply:

```
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.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

```
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.

Logger Name	Assigned LoggerConfig	LoggerConfig Level	Logger Level
root	root	DEBUG	DEBUG
X	root	DEBUG	DEBUG
X.Y	root	DEBUG	DEBUG
X.Y.Z	root	DEBUG	DEBUG

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.

Logger Name	Assigned LoggerConfig	LoggerConfig Level	Level
root	root	DEBUG	DEBUG
X	X	ERROR	ERROR
X.Y	X.Y	INFO	INFO
X.Y.Z	X.Y.Z	WARN	WARN

Example 2

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

Logger Name	Assigned LoggerConfig	LoggerConfig Level	Level
root	root	DEBUG	DEBUG
X	X	ERROR	ERROR
X.Y	X	ERROR	ERROR
X.Y.Z	X.Y.Z	WARN	WARN

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.

Logger Name	Assigned LoggerConfig	LoggerConfig Level	level
root	root	DEBUG	DEBUG
X	X	ERROR	ERROR
X.Y	X	ERROR	ERROR
X.Y.Z	X	ERROR	ERROR

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.

Logger Name	Assigned LoggerConfig	LoggerConfig Level	level
root	root	DEBUG	DEBUG
X	X	ERROR	ERROR
X.Y	X.Y	INFO	INFO
X.YZ	X	ERROR	ERROR

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.

Logger Name	Assigned LoggerConfig	LoggerConfig Level	Level
root	root	DEBUG	DEBUG
X	X	ERROR	ERROR
X.Y	X.Y		ERROR
X.Y.Z	X.Y		ERROR

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).

Event Level	LoggerCon Level						
	TRACE	DEBUG	INFO	WARN	ERROR	FATAL	OFF
ALL	YES	YES	YES	YES	YES	YES	NO
TRACE	YES	NO	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO	NO
ERROR	YES	YES	YES	YES	YES	NO	NO
FATAL	YES	YES	YES	YES	YES	YES	NO
OFF	NO	NO	NO	NO	NO	NO	NO

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:

Logger Name	Added Appenders	Additivity Flag	Output Targets	Comment
root	A1	not applicable	A1	The root logger has no parent so additivity does not apply to it.
x	A-x1, A-x2	true	A1, A-x1, A-x2	Appenders of "x" and root.
x.y	none	true	A1, A-x1, A-x2	Appenders of "x" and root. It would not be typical to configure a Logger with no Appenders.
x.y.z	A-xyz1	true	A1, A-x1, A-x2, A-xyz1	Appenders in "x.y.z", "x" and root.

security	A-sec	false	A-sec	No appender accumulation since the additivity flag is set to `false`.
security.access	none	true	A-sec	Only appenders of "security" because the additivity flag in "security" is set to `false`.

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:

```
176 [main] INFO  org.foo.Bar - Located nearest gas station.
```

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.
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 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. 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 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 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

Log4j 1.x XML configuration

```
<?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="A1" class="org.apache.log4j.FileAppender">
    <param name="File"   value="A1.log" />
    <param name="Append" value="false" />
    <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

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <File name="A1" fileName="A1.log" append="false">
      <PatternLayout pattern="%t %-5p %c{2} - %m%n"/>
    </File>
    <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="debug">
      <AppenderRef ref="A1"/>
    </Logger>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</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"?>
<!DOCTYPE log4j:configuration PUBLIC "-//APACHE//DTD LOG4J 1.2//EN" "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <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

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Socket name="A1" host="localHost" port="5000">
      <SerializedLayout/>
    </Socket>
    <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="debug">
      <AppenderRef ref="A1"/>
    </Logger>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

3.1.3.4 Sample 4 - AsyncAppender

Log4j 1.x XML configuration using the AsyncAppender.

```
<?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/" 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.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
    </layout>
  </appender>
  <root>
    <priority value="debug"/>
    <appender-ref ref="ASYNC"/>
  </Root>
</log4j:configuration>
```

Log4j 2 XML configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug">
  <Appenders>
    <File name="TEMP" fileName="temp">
      <PatternLayout pattern="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
    </File>
    <Async name="ASYNC">
      <AppenderRef ref="TEMP"/>
    </Async>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="ASYNC"/>
    </Root>
  </Loggers>
</Configuration>
```

3.1.3.5 Sample 5 - AsyncAppender with Console and File

Log4j 1.x XML configuration using the AsyncAppender.

```
<?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/" configDebug="true">
  <appender name="ASYNC" class="org.apache.log4j.AsyncAppender">
    <appender-ref ref="TEMP"/>
    <appender-ref ref="CONSOLE"/>
  </appender>
  <appender name="CONSOLE" 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>
  <appender name="TEMP" class="org.apache.log4j.FileAppender">
    <param name="File" value="temp"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
    </layout>
  </appender>
  <root>
    <priority value="debug"/>
    <appender-ref ref="ASYNC"/>
  </Root>
</log4j: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 API

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.

```
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:

```
if (logger.isDebugEnabled()) {
    logger.debug("Logging in user " + user.getName() + " with birthday " + 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:

```
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

Substituting parameters leaves 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 method. The output for this example shows that Calendar toString() is verbose compared to custom formatting:

```
2012-12-12 11:56:19,633 [main] DEBUG: User John Smith with birthday java.util.GregorianCalendar[time=?,areFie
2012-12-12 11:56:19,643 [main] DEBUG: User John Smith with birthday 05 23, 1995
2012-12-12 11:56:19,643 [main] DEBUG: Integer.MAX_VALUE = 2,147,483,647
2012-12-12 11:56:19,643 [main] DEBUG: Long.MAX_VALUE = 9,223,372,036,854,775,807
```

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.getBir
```

4.1.1.5 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.appender 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 both examples below the Logger will have a name of "org.apache.test.MyTest".

```
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, or YAML.
2. Programmatically, by creating a ConfigurationFactory and Configuration implementation.
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.

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 then in weighted order from highest to lowest. As delivered, Log4j contains three ConfigurationFactory implementations: one for JSON, one for YAML, 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 YAML ConfigurationFactory will look for `log4j2-test.yaml` or `log4j2-test.yml` in the classpath.
3. If no such file is found the JSON ConfigurationFactory will look for `log4j2-test.json` or `log4j2-test.jsn` in the classpath.
4. If no such file is found the XML ConfigurationFactory will look for `log4j2-test.xml` in the classpath.
5. If a test file cannot be located the YAML ConfigurationFactory will look for `log4j2.yaml` or `log4j2.yml` on the classpath.
6. If a YAML file cannot be located the JSON ConfigurationFactory will look for `log4j2.json` or `log4j2.jsn` on the classpath.
7. If a JSON file cannot be located the XML ConfigurationFactory will try to locate `log4j2.xml` on the classpath.
8. 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.

```
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`.

```
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 com.foo.Bar - Did it again!
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 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] TRACE MyApp - Entering application.
17:13:01.540 [main] TRACE com.foo.Bar - entry
17:13:01.540 [main] ERROR com.foo.Bar - Did it again!
17:13:01.540 [main] TRACE com.foo.Bar - exit with (false)
17:13:01.540 [main] ERROR MyApp - Didn't do it.
17:13:01.540 [main] TRACE MyApp - Exiting application.
```

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

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:

```
<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.

5.1.2 Additivity

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 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] TRACE com.foo.Bar - entry
17:13:01.540 [main] TRACE com.foo.Bar - entry
17:13:01.540 [main] ERROR com.foo.Bar - Did it again!
17:13:01.540 [main] TRACE com.foo.Bar - exit (false)
17:13:01.540 [main] TRACE com.foo.Bar - exit (false)
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` is first used, which 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 is 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 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" advert
  ...
  </File>
</Appenders>
```

5.1.5 Configuration Syntax

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.

5.1.5.1 Configuration with XML

The configuration element in the XML file accepts several attributes:

Attribute Name	Description
advertiser	(Optional) The Advertiser plugin name which will be used to advertise individual FileAppender or SocketAppender configurations. The only Advertiser plugin provided is 'multicastdns''.
dest	Either "err", which will send output to stderr, or a file path or URL.
monitorInterval	The minimum amount of time, in seconds, that must elapse before the file configuration is checked for changes.
name	The name of the configuration.
packages	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.
schema	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.
shutdownHook	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 "disable"
status	The level of internal Log4j events that should be logged to the console. Valid values for this attribute are "trace", "debug", "info", "warn", "error" and "fatal". Log4j will log details about initialization, rollover and other internal actions to the status logger. Setting `status="trace"` is one of the first tools available to you if you need to troubleshoot log4j.
strict	Enables the use of the strict XML format. Not supported in JSON configurations.
verbose	Enables diagnostic information while loading plugins.

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

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

and

```
<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 version="1.0" encoding="UTF-8"?>;
<Configuration>
  <Properties>
    <Property name="name1">value</property>
    <Property name="name2" value="value2"/>
  </Properties>
  <
filter  ... />
  <Appenders>
    <
appender ... >
      <
filter  ... />
    </
appender>
    ...
  </Appenders>
  <Loggers>
    <Logger name="name1">
      <
filter  ... />
    </Logger>
    ...
    <Root level="level">
      <AppenderRef ref="name"/>
    </Root>
  </Loggers>
</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 configuerd using an element named Console it is instead configured as an appender element with a type attribute containing "Console".

```
<?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>
  <Loggers>
    <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>
    <Appender type="List" name="List">
    </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="List"/>
    </Root>
  </Loggers>

</Configuration>
```

5.1.5.2 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.

```
{ "configuration": { "status": "error", "name": "RoutingTest",
                     "packages": "org.apache.logging.log4j.test",
      "properties": {
        "property": { "name": "filename",
                      "value" : "target/rolling1/rollingtest-$${sd:type}.log" }
      },
    "ThresholdFilter": { "level": "debug" },
    "appenders": {
      "Console": { "name": "STDOUT",
        "PatternLayout": { "pattern": "%m%n" }
      },
      "List": { "name": "List",
        "ThresholdFilter": { "level": "debug" }
      },
      "Routing": { "name": "Routing",
        "Routes": { "pattern": "$${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"},
            { "AppenderRef": "List", "key": "Service"}
          ]
        }
      }
    },
    "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-$${sd:type}.log" }
      },
    "ThresholdFilter": { "level": "debug" },
    "appenders": {
      "appender": [
         { "type": "Console", "name": "STDOUT", "PatternLayout": { "pattern": "%m%n" }},
         { "type": "List", "name": "List", "ThresholdFilter": { "level": "debug" }},
         { "type": "Routing",  "name": "Routing",
          "Routes": { "pattern": "$${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"},
              { "AppenderRef": "List", "key": "Service"}
            ]
          }
        }
      ]
    },
    "loggers": {
      "logger": [
        { "name": "EventLogger", "level": "info", "additivity": "false",
          "AppenderRef": { "ref": "Routing" }},
        { "name": "com.foo.bar", "level": "error", "additivity": "false",
          "AppenderRef": { "ref": "Console" }}
      ],
      "root": { "level": "error", "AppenderRef": { "ref": "STDOUT" }}
    }
  }
}
```

The JSON support uses the Jackson Data Processor to parse the JSON files. These dependencies must be added to a project that wants to use JSON for configuration:

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>${jackson2Version}</version>
</dependency>

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>${jackson2Version}</version>
</dependency>

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>${jackson2Version}</version>
</dependency>
```

5.1.5.3 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 false 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 ThreadContext Map. 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.

5.1.5.4 Configuring Appenders

An appender is configured either using the specific appender plugin's name or with an appender element and the type attibute 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.

5.1.5.5 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>
    <List name="List">
    </List>
  </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="List"/>
    </Root>
  </Loggers>

</Configuration>
```

5.1.6 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 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"/>
    </Console>
    <List name="List">
      <ThresholdFilter level="debug"/>
    </List>
    <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"/>
        <Route ref="List" key="Service"/>
      </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. The contexts that are built in to Logj4 are:

Prefix	Context

bundle	Resource bundle. The format is `${bundle:BundleName:BundleKey}`. The bundle name follows package naming conventions, for example: `${bundle:com.domain.Messages:MyKey}`.
ctx	Thread Context Map (MDC)
date	Inserts the current date and/or time using the specified format
env	System environment variables
jvmrunargs	A JVM input argument accessed through JMX, but not a main argument; see RuntimeMXBean.getInputArguments(). Not available on Android.
main	A value set with MapLookup.setMainArguments(String[])
map	A value from a MapMessage
sd	A value from a StructuredDataMessage. The key "id" will return the name of the StructuredDataId without the enterprise number. The key "type" will return the message type. Other keys will retrieve individual elements from the Map.
sys	System properties

A default property map can be declared in the configuration file. 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.

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 '$' characters. When the configuration file is first processed the first variable is simply removed. Thus, when the Routes element is evaluated at runtime it is the variable declaration "${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 version="1.0" encoding="UTF-8"?>
<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.

5.1.7 XInclude

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

```
<?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 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 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>
```

5.1.8 Status Messages

Troubleshooting tip for the impatient:

- 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 accomodate 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 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"/>
    </Console>
    <List name="List">
      <ThresholdFilter level="debug"/>
    </List>
    <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"/>
        <Route ref="List" key="Service"/>
      </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 17:08:00,800 DEBUG Calling createLayout on class org.apache.logging.log4j.core.
       layout.PatternLayout for element PatternLayout with params(pattern="%m%n",
       Configuration(RoutingTest), null, charset="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 17:08:00,806 DEBUG Calling createAppender on class org.apache.logging.log4j.test.
       appender.ListAppender for element List with params(name="List", entryPerNewLine="null",
       raw="null", null, ThresholdFilter(DEBUG))
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 17:08:00,824 DEBUG Calling createRoute on class org.apache.logging.log4j.core.appender.
       routing.Route for element Route with params(AppenderRef="List", key="Service", 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'),
       Route(type=static Reference=List key='Service')})
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'),
       Route(type=static Reference=List key='Service')}), 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, List, 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")
```

```
2011-11-23 17:08:00,831 DEBUG Calling createLogger on class org.apache.logging.log4j.core.config.
       LoggerConfig$RootLogger for element root with params(additivity="null", level="error",
       AppenderRef={STDOUT}, null)
2011-11-23 17:08:00,833 DEBUG Calling createLoggers on class org.apache.logging.log4j.core.
       config.plugins.LoggersPlugin for element loggers with params(loggers={EventLogger, root})
2011-11-23 17:08:00,834 DEBUG Reconfiguration completed
2011-11-23 17:08:00,846 DEBUG Calling createLayout on class org.apache.logging.log4j.core.
       layout.PatternLayout for element PatternLayout with params(pattern="%d %p %c{1.} [%t] %m%n",
       Configuration(RoutingTest), null, charset="null")
2011-11-23 17:08:00,849 DEBUG Calling createPolicy on class org.apache.logging.log4j.core.
       appender.rolling.SizeBasedTriggeringPolicy for element SizeBasedTriggeringPolicy with
       params(size="500")
2011-11-23 17:08:00,851 DEBUG Calling createAppender on class org.apache.logging.log4j.core.
       appender.RollingFileAppender for element RollingFile with
       params(fileName="target/rolling1/rollingtest-Unknown.log",
       filePattern="target/rolling1/test1-Unknown.%i.log.gz", append="null", name="Rolling-Unknown",
       bufferedIO="null", immediateFlush="null",
       SizeBasedTriggeringPolicy(SizeBasedTriggeringPolicy(size=500)), null,
       PatternLayout(%d %p %c{1.} [%t] %m%n), null, ignoreExceptions="null")
2011-11-23 17:08:00,858 DEBUG Generated plugins in 0.002014000 seconds
2011-11-23 17:08:00,889 DEBUG Reconfiguration started for context sun.misc.
       Launcher$AppClassLoader@37b90b39
2011-11-23 17:08:00,890 DEBUG Generated plugins in 0.001355000 seconds
2011-11-23 17:08:00,959 DEBUG Generated plugins in 0.001239000 seconds
2011-11-23 17:08:00,961 DEBUG Generated plugins in 0.001197000 seconds
2011-11-23 17:08:00,965 WARN No Loggers were configured, using default
2011-11-23 17:08:00,976 DEBUG Reconfiguration completed
```

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:

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

the following error message will be produced.

```
2011-11-24 23:21:25,517 ERROR Unable to locate appender Routng for logger EventLogger
```

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:

```
StatusConsoleListener listener = new StatusConsoleListener(Level.ERROR);
StatusLogger.getLogger().registerListener(listener);
```

5.1.9 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 `InitialLoggerContext` 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:

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

    @Test
    public void testSomeAwesomeFeature() {
        final LoggerContext ctx = init.getContext();
        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.
    }
}
```

5.1.10 System Properties

Below follows a number of system properties that can be used to control Log4j 2 behaviour. Any spaces present in the property name are for visual flow and should be removed.

System Property	Default Value	Description
log4j.configurationFile		Path to an XML or JSON Log4j 2 configuration file.

Log4jContextSelector	ClassLoaderContextSelector	Creates the `LoggerContexts`. An application can have one or more active LoggerContexts depending on the circumstances. See Log Separation for more details. Available context selector implementation classes: `org.apache.logging.log4j.core.async .` - makes all loggers asynchronous. `org.apache.logging.log4j.core.selecto` - creates a single shared LoggerContext. `org.apache.logging.log4j.core.selecto` - separate LoggerContexts for each web application. `org.apache.logging.log4j.core.selecto` - use JNDI to locate each web application's LoggerContext. `org.apache.logging.log4j.core.osgi .B` - separate LoggerContexts for each OSGi bundle.
Log4jLogEventFactory	org.apache.logging.log4j.core.impl .D	Factory class used by LoggerConfig to create `LogEvent` instances. (Ignored when the `AsyncLoggerContextSelector` is used.)
log4j2.loggerContextFactory	org.apache.logging.log4j.simple .Sim	Factory class used by LogManager to bootstrap the logging implementation. The core jar provides `org.apache.logging.log4j.core.impl.Lo`
log4j.configurationFactory		Fully specified class name of a class extending `org.apache.logging.log4j.core.config.` If specified, an instance of this class is added to the list of configuration factories.
log4j.shutdownHookEnabled	true	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.
log4j.shutdownCallbackRegistry	org.apache.logging.log4j.core.util .De	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.

log4j.Clock	SystemClock	Implementation of the `org.apache.logging.log4j.core.util.Cl` 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.
org.apache.logging.log4j.level	ERROR	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).
log4j2.enableJndiContextSelector	false	When true, the Log4j context selector that uses the JNDI java protocol is enabled. When false, the default, they are disabled.
log4j2.enableJndiJdbc	false	When true, a Log4j JDBC Appender configured with a `DataSource` which uses JNDI's java protocol is enabled. When false, the default, they are disabled.
log4j2.enableJndiJms	false	When true, a Log4j JMS Appender that uses JNDI's java protocol is enabled. When false, the default, they are disabled.
log4j2.enableJndiLookup	false	When true, a Log4j lookup that uses JNDI's java protocol is enabled. When false, the default, they are disabled.
disableThreadContext	false	If `true`, the ThreadContext stack and map are disabled. (May be ignored if a custom ThreadContext map is specified.)
disableThreadContextStack	false	If `true`, the ThreadContext stack is disabled.
disableThreadContextMap	false	If `true`, the ThreadContext map is disabled. (May be ignored if a custom ThreadContext map is specified.)
log4j2.threadContextMap		Fully specified class name of a custom `ThreadContextMap` implementation class.
isThreadContextMapInheritable	false	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.)

log4j2.disable.jmx	false	If `true`, Log4j configuration objects like LoggerContexts, Appenders, Loggers, etc. will not be instrumented with MBeans and cannot be remotely monitored and managed.
log4j2.jmx.notify.async	false for web apps, true otherwise	If `true`, log4j's JMX notifications are sent from a separate background thread, otherwise they are sent from the caller thread. If the `javax.servlet.Servlet` class is on the classpath, the default behaviour is to use the caller thread to send JMX notifications.
log4j.skipJansi	false	If `true`, the ConsoleAppender will not try to use the Jansi output stream on Windows.
log4j.ignoreTCL	false	If `true`, classes are only loaded with the default class loader. Otherwise, an attempt is made to load classes with the current thread's context class loader before falling back to the default class loader.
org.apache.logging.log4j.uuidSequen	0	System property that may be used to seed the UUID generation with an integer value.
org.apache.logging.log4j.simplelog .s	false	If `true`, the full ThreadContext map is included in each SimpleLogger log message.
org.apache.logging.log4j.simplelog .s	false	If `true`, the logger name is included in each SimpleLogger log message.
org.apache.logging.log4j.simplelog .s	true	If `true`, only the last component of a logger name is included in SimpleLogger log messages. (E.g., if the logger name is "mycompany.myproject.mycomponent", only "mycomponent" is logged.
org.apache.logging.log4j.simplelog .s	false	If `true`, SimpleLogger log messages contain timestamp information.
org.apache.logging.log4j.simplelog .d	"yyyy/MM/dd HH:mm:ss:SSS zzz"	Date-time format to use. Ignored if `org.apache.logging.log4j.simplelog.sh` is `false`.
org.apache.logging.logj.simplelog .log	system.err	"system.err" (case-insensitive) logs to System.err, "system.out" (case-insensitive) logs to System.out, any other value is interpreted as a file name to save SimpleLogger messages to.

org.apache.logging.log4j.simplelog .le	ERROR	Default level for new SimpleLogger instances.
org.apache.logging.log4j.simplelog.<l	SimpleLogger default log level	Log level for a the SimpleLogger instance with the specified name.
org.apache.logging.log4j.simplelog .S	ERROR	This property is used to control the initial StatusLogger level, and can be overridden in code by calling `StatusLogger.getLogger().setLevel(som` 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).
Log4jDefaultStatusLevel	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.

log4j2.StatusLogger.level	WARN	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 .S`
log4j2.status.entries	200	Number of StatusLogger events that are kept in a buffer and can be retrieved with `StatusLogger.getStatusData()`.
AsyncLogger.ExceptionHandler		See Async Logger System Properties for details.
AsyncLogger.RingBufferSize	256 * 1024	See Async Logger System Properties for details.
AsyncLogger.WaitStrategy	Sleep	See Async Logger System Properties for details.
AsyncLogger.ThreadNameStrategy	CACHED	See Async Logger System Properties for details.
AsyncLoggerConfig.ExceptionHandle		See Mixed Async/Synchronous Logger System Properties for details.
AsyncLoggerConfig.RingBufferSize	256 * 1024	See Mixed Async/Synchronous Logger System Properties for details.

AsyncLoggerConfig.WaitStrategy	Sleep	See Mixed Async/Synchronous Logger System Properties for details.
log4j.jul.LoggerAdapter	org.apache.logging.log4j.jul .ApiLogg	Default LoggerAdapter to use in the JUL adapter. By default, if log4j-core is available, then the class `org.apache.logging.log4j.jul .CoreLog` will be used. Otherwise, the `ApiLogggerAdapter` will be used. Custom implementations must provide a public default constructor.

Log4j 2 System 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 in programmatically, it will be too late for Log4j to detect the setting.

```
<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"

```
<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`

```
<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 ServletExceptio
        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 ServletExcepti
        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, IOExcepti
        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, IOExcepti
         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.

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 of goals 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 four places:

- Serialized plugin listing files on the classpath. These files are generated automatically during the build (more details below).
- (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.
- A comma-separated list of packages specified by the `log4j.plugin.packages` system property.
- Packages passed to the static `PluginManager.addPackages` method (before Log4j configuration occurs).
- 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</annota
        </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, 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 that is marked with a PluginFactory annotation. 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.

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.

PluginElement

The parameter may represent a complex object that itself has parameters that can be configured.

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.

Required

While not strictly an attribute, this annotation can be added to any plugin factory parameter to make it automatically validated as non- `null` and non-empty.

7.1.3 Converters

Converters are used by PatternLayout to render the elements identified by the conversion pattern. Every converter must specify its type as "Converter" on the Plugin attribute, 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 type 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.

```
<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.

```
<RollingFile name="Rolling-${map:type}" fileName="${filename}" filePattern="target/rolling1/test1-$${date:MM-
  <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.

```
<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.

Key	Description
version	The short Java version, like: `Java version 1.7.0_67`
runtime	The Java runtime version, like: `Java(TM) SE Runtime Environment (build 1.7.0_67-b01) from Oracle Corporation`
vm	The Java VM version, like: `Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)`
os	The OS version, like: `Windows 7 6.1 Service Pack 1, architecture: amd64-64`
locale	Hardware information, like: `default locale: en_US, platform encoding: Cp1252`
hw	Hardware information, like: `processors: 4, architecture: amd64-64, instruction sets: amd64`

For example:

```
<File name="Application" fileName="application.log">
  <PatternLayout header="${java:runtime} - ${java:vm} - ${java:os}">
    <Pattern>%d %m%n</Pattern>
  </PatternLayout>
</File>
```

8.1.5 Jndi Lookup

As of Log4j 2.13.1 JNDI operations require that `log4j2.enableJndiLookup=true` be set as a system property or the corresponding environment variable for this lookup to function. See the enableJndiLookup system property.

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.

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

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.

8.1.7 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.MapLookup;

public static void main(String args[]) {
  MapLookup.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 `${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:

Expression	Result
${main:0}	`--file`
${main:1}	`foo.txt`
${main:2}	`--verbose`

${main:3}	-x
${main:4}	bar
${main:--file}	foo.txt
${main:-x}	bar
${main:bar}	null

Example usage:

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

8.1.8 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. `${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.

```
<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>
```

8.1.9 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.

```
<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>
```

8.1.10 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:

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

8.1.11 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:

Key	Description
attr. *name*	Returns the ServletContext attribute with the specified name
contextPath	The context path of the web 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.

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.

Parameter Name	Type	Description
AppenderRef	String	The name of the Appenders to invoke asynchronously. Multiple AppenderRef elements can be configured.
blocking	boolean	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.
bufferSize	integer	Specifies the maximum number of events that can be queued. The default is 128.
errorRef	String	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.
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.
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.
includeLocation	boolean	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 includeLocation="true".

AsyncAppender Parameters

A typical AsyncAppender configuration might look like:

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

9.1.2 ConsoleAppender

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

Parameter Name	Type	Description
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.

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.
follow	boolean	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.
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.
target	String	Either "SYSTEM_OUT" or "SYSTEM_ERR". The default is "SYSTEM_ERR".

ConsoleAppender Parameters

A typical Console configuration might look like:

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

9.1.3 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

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.
primary	String	The name of the primary Appender to use.
failovers	String[]	The names of the secondary Appenders to use.
name	String	The name of the Appender.
retryIntervalSeconds	integer	The number of seconds that should pass before retrying the primary Appender. The default is 60.
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.
target	String	Either "SYSTEM_OUT" or "SYSTEM_ERR". The default is "SYSTEM_ERR".

FailoverAppender Parameters

A Failover configuration might look like:

```
<?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"
                 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>
```

9.1.4 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.

Parameter Name	Type	Description
append	boolean	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.
bufferedIO	boolean	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.
bufferSize	int	When bufferedIO is true, this is the buffer size, the default is 8192 bytes.
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.
fileName	String	The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.
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.

layout	Layout	The Layout to use to format the LogEvent
locking	boolean	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 "advisory" meaning that other applications can perform operations on the file without acquiring a lock. The default value is false.
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.

FileAppender Parameters

Here is a sample File configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="MyFile"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.5 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.

Parameter Name	Type	Description
agents	Agent[]	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.
agentRetries	integer	The number of times the agent should be retried before failing to a secondary. This parameter is ignored when type="persistent" is specified (agents are tried once before failing to the next).
batchSize	integer	Specifies the number of events that should be sent as a batch. The default is 1. *This parameter only applies to the Flume Appender.*
compress	boolean	When set to true the message body will be compressed using gzip
connectTimeoutMillis	integer	The number of milliseconds Flume will wait before timing out the connection.
dataDir	String	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.

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.
eventPrefix	String	The character string to prepend to each event attribute in order to distinguish it from MDC attributes. The default is an empty string.
flumeEventFactory	FlumeEventFactory	Factory that generates the Flume events from Log4j events. The default factory is the FlumeAvroAppender itself.
layout	Layout	The Layout to use to format the LogEvent. If no layout is specified RFC5424Layout will be used.
lockTimeoutRetries	integer	The number of times to retry if a LockConflictException occurs while writing to Berkeley DB. The default is 5.
maxDelayMillis	integer	The maximum number of milliseconds to wait for batchSize events before publishing the batch.
mdcExcludes	String	A comma separated list of mdc keys that should be excluded from the FlumeEvent. This is mutually exclusive with the mdcIncludes attribute.
mdcIncludes	String	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.
mdcRequired	String	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.
mdcPrefix	String	A string that should be prepended to each MDC key in order to distinguish it from event attributes. The default string is "mdc:".
name	String	The name of the Appender.

properties	Property[]	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 "sources.log4j-source.interceptors". 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: 1. "keyProvider" to specify the name of the plugin to provide the secret key for encryption.
requestTimeoutMillis	integer	The number of milliseconds Flume will wait before timing out the request.
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.
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 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 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 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>
```

9.1.6 JDBCAppender

The JDBC Appender 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.

The JDBC Appender configured with a `DataSource` requires JNDI support so as of release 2.17.1 this appender will not function unless `log4j2.enableJndiJdbc=true` is configured as a system property or environment variable. See the enableJndiJdbc system property.

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.

Parameter Name	Type	Description
name	String	*Required.* 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.
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.
connectionSource	ConnectionSource	*Required.* The connections source from which database connections should be retrieved.
tableName	String	*Required.* The name of the database table to insert log events into.
columnConfigs	ColumnConfig[]	*Required.* Information about the columns that log event data should be inserted into and how to insert that data. This is represented with multiple `<Column>` elements.

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 `<DataSource>` or `<ConnectionFactory>` nested elements.

Parameter Name	Type	Description

jndiName	String	*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.

DataSource Parameters

Parameter Name	Type	Description
class	Class	*Required.* The fully qualified name of a class containing a static factory method for obtaining JDBC connections.
method	Method	*Required.* The name of a static factory method for obtaining JDBC connections. This method must have no parameters and its return type must be either `java.sql.Connection` or `DataSource`. If the method returns `Connection`s, it must obtain them from a connection pool (and they will be returned to the pool when Log4j is done with them); otherwise, logging will be very slow. If the method returns a `DataSource`, the `DataSource` will only be retrieved once, and it must be backed by a connection pool for the same reasons.

ConnectionFactory Parameters

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.

Parameter Name	Type	Description
name	String	*Required.* The name of the database column.
pattern	String	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="true"` must be specified, but not more than one of these.

literal	String	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="'Literal String'"`). This is especially useful for databases that don't support identity columns. For example, if you are using Oracle you could specify `literal="NAME_OF_YOUR_SEQUENCE.NEXTVA` to insert a unique ID in an ID column. Either this attribute, `pattern`, or `isEventTimestamp="true"` must be specified, but not more than one of these.
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`).

Column 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_COMMI
        );

        this.dataSource = new PoolingDataSource(pool);
    }

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

9.1.7 JMSAppender

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

The JMS Appender requires JNDI support so as of release 2.3.1 this appender will not function unless `log4j2.enableJndiJms=true` is configured as a system property or environment variable. See the enableJndiJms system property.

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 JMSAppender 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.

Parameter Name	Type	Description
factoryBindingName	String	The name to locate in the Context that provides the ConnectionFactory. This can be any subinterface of `ConnectionFactory` as well. This attribute is required.
factoryName	String	The fully qualified class name that should be used to define the Initial Context Factory as defined in INITIAL_CONTEXT_FACTORY. If no value is provided the default InitialContextFactory will be used. If a factoryName is specified without a providerURL a warning message will be logged as this is likely to cause problems.
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.
layout	Layout	The Layout to use to format the LogEvent. If you do not specify a layout, this appender will use a SerializedLayout.
name	String	The name of the Appender. Required.
password	String	The password to use to create the JMS connection.
providerURL	String	The URL of the provider to use as defined by PROVIDER_URL. If this value is null the default system provider will be used.
destinationBindingName	String	The name to use to locate the Destination. This can be a `Queue` or `Topic`, and as such, the attribute names `queueBindingName` and `topicBindingName` are aliases to maintain compatibility with the Log4j 2.0 JMS appenders.
securityPrincipalName	String	The name of the identity of the Principal as specified by SECURITY_PRINCIPAL. If a securityPrincipalName is specified without securityCredentials a warning message will be logged as this is likely to cause problems.

securityCredentials	String	The security credentials for the principal as specified by SECURITY_CREDENTIALS.
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.
urlPkgPrefixes	String	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.
userName	String	The user id used to create the JMS connection.

JMSAppender Parameters

Here is a sample JMSAppender configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp">
  <Appenders>
    <JMS name="jmsQueue" destinationBindingName="MyQueue"
         factoryBindingName="MyQueueConnectionFactory"/>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="jmsQueue"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.8 JPAAppender

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.

Parameter Name	Type	Description
name	String	*Required.* 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.
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.
entityClassName	String	*Required.* The fully qualified name of the concrete LogEventWrapperEntity implementation that has JPA annotations mapping it to a database table.
persistenceUnitName	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 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>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                                 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             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();
    }
    ...
}
```

9.1.9 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 MemoryMappedFileManagers 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.

Parameter Name	Type	Description
append	boolean	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.
fileName	String	The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.
filters	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.

immediateFlush	boolean	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. 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. Note that manually forcing a sync on every log event loses most of the performance benefits of using a memory mapped file. 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.
regionLength	int	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.
layout	Layout	The Layout to use to format the LogEvent
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.

MemoryMappedFileAppender Parameters

Here is a sample MemoryMappedFile configuration:

```
<?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>
```

9.1.10 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.

Parameter Name	Type	Description
name	String	*Required.* 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.
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.
NoSqlProvider	NoSQLProvider<C extends NoSQLConnection<W, T extends NoSQLObject<W>>>	*Required.* The NoSQL provider that provides connections to the chosen NoSQL database.

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.

Parameter Name	Type	Description
collectionName	String	*Required.* The name of the MongoDB collection to insert the events into.
writeConcernConstant	Field	By default, the MongoDB provider inserts records with the instructions `com.mongodb.WriteConcern.ACKNOWLEDGED`. Use this optional attribute to specify the name of a constant other than `ACKNOWLEDGED`.
writeConcernConstantClass	Class	If you specify `writeConcernConstant`, you can use this attribute to specify a class other than `com.mongodb.WriteConcern` to find the constant on (to create your own custom instructions).
factoryClassName	Class	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.DB` or a `com.mongodb.MongoClient`. 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 `databaseName`, `server`, or `port` attributes.
factoryMethodName	Method	See the documentation for attribute `factoryClassName`.
databaseName	String	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).
server	String	See the documentation for attribute `databaseName`.
port	int	See the documentation for attribute `databaseName`.

username	String	See the documentation for attributes `databaseName` and `factoryClassName`.
password	String	See the documentation for attributes `databaseName` and `factoryClassName`.

MongoDB Provider Parameters

Parameter Name	Type	Description
factoryClassName	Class	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.
factoryMethodName	Method	See the documentation for attribute `factoryClassName`.
databaseName	String	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).
protocol	String	Must either be "http" or "https." See the documentation for attribute `databaseName`.
server	String	See the documentation for attribute `databaseName`.
port	int	See the documentation for attribute `databaseName`.
username	String	See the documentation for attributes `databaseName`.

password	String	See the documentation for attributes `databaseName`.

CouchDB Provider Parameters

Here are a few sample configurations for the NoSQLAppender:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <NoSql name="databaseAppender">
      <MongoDb 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 version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <NoSql name="databaseAppender">
      <MongoDb collectionName="applicationLog" factoryClassName="org.example.db.ConnectionFactory"
               factoryMethodName="getNewMongoClient" />
    </NoSql>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <NoSql name="databaseAppender">
      <CouchDb databaseName="applicationDb" protocol="https" server="couch.example.org"
               username="loggingUser" password="abc123" />
    </NoSql>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="databaseAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

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": 1368844166761,
    "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", "file
                { "className": "org.example.sql.driver.PreparedStatement", "methodName": "executeUpdate", "fi
                { "className": "com.example.application.MyClass", "methodName": "exampleMethod", "fileName":
                { "className": "com.example.application.MainClass", "methodName": "main", "fileName": "MainCl
        ],
        "cause": {
            "type": "java.io.IOException",
            "message": "Connection lost.",
            "stackTrace": [
                { "className": "java.nio.channels.SocketChannel", "methodName": "write", "fileName": null, "l
                { "className": "org.example.sql.driver.PreparedStatement$1", "methodName": "responder", "file
                { "className": "org.example.sql.driver.PreparedStatement", "methodName": "executeUpdate", "fi
                { "className": "com.example.application.MyClass", "methodName": "exampleMethod", "fileName":
                { "className": "com.example.application.MainClass", "methodName": "main", "fileName": "MainCl
            ]
        }
    },
    "contextMap": {
        "ID": "86c3a497-4e67-4eed-9d6a-2e5797324d7b",
        "username": "JohnDoe"
    },
    "contextStack": [
        "topItem",
        "anotherItem",
        "bottomItem"
    ]
}
```

9.1.11 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.

9.1.12 RandomAccessFileAppender

As of beta-9, the name of this appender has been changed from FastFile to RandomAccessFile. Configurations using the `FastFile` element no longer work and should be modified to use the `RandomAccessFile` element.

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 RandomAccessFileManagers 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.

Parameter Name	Type	Description
append	boolean	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.
fileName	String	The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.
filters	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.

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
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 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>
```

9.1.13 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.

Parameter Name	Type	Description
AppenderRef	String	The name of the Appenders to call after the LogEvent has been manipulated. Multiple AppenderRef elements can be configured.
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.
name	String	The name of the Appender.
rewritePolicy	RewritePolicy	The RewritePolicy that will manipulate the LogEvent.
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.

RewriteAppender Parameters

9.1.13.1 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.

Parameter Name	Type	Description
mode	String	"Add" or "Update"
keyValuePair	KeyValuePair[]	An array of keys and their values.

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"/>
      <MapRewritePolicy mode="Add">
        <KeyValuePair key="product" value="TestProduct"/>
      </MapRewritePolicy>
    </Rewrite>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Rewrite"/>
    </Root>
  </Loggers>
</Configuration>
```

9.PropertiesRewritePolicy

PropertiesRewritePolicy 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.

Parameter Name	Type	Description
properties	Property[]	One of more Property elements to define the keys and values to be added to the ThreadContext Map.

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>
```

9.1.14 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 RollingFileManagers 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.

File locking is not supported by the RollingFileAppender.

Parameter Name	Type	Description
append	boolean	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.

bufferedIO	boolean	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.
bufferSize	int	When bufferedIO is true, this is the buffer size, the default is 8192 bytes.
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.
fileName	String	The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.
filePattern	String	The pattern of the file name of the archived log file. The format of the pattern should is dependent on the RolloverPolicy that is used. The DefaultRolloverPolicy will accept both a date/time pattern compatible with SimpleDateFormat and 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.
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.
layout	Layout	The Layout to use to format the LogEvent

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.

RollingFileAppender Parameters

9.1.14.1 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.

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

9.OnStartup Triggering Policy

The `OnStartupTriggeringPolicy` policy takes no parameters and causes a rollover if the log file is older than the current JVM's start time.

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`.

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 `increment` attribute which indicates how frequently the rollover should occur based on the time pattern and a `modulate` boolean attribute.

Parameter Name	Type	Description
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 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 then the first rollover will occur at 4 am and then next ones will occur at 8 am, noon, 4pm, etc.

TimeBasedTriggeringPolicy Parameters

9.1.14.2 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" or ".zip" the resulting archive will be compressed using the compression scheme that matches the suffix. 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 two 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".

Number of rollovers	Active output target	Archived log files	Description
0	foo.log	-	All logging is going to the initial file.

1	foo.log	foo-1.log	During the first rollover foo.log is renamed to foo-1.log. A new foo.log file is created and starts being written to.
2	foo.log	foo-1.log, foo-2.log	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.
3	foo.log	foo-1.log, foo-2.log, foo-3.log	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.
4	foo.log	foo-1.log, foo-2.log, foo-3.log	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.

By way of contrast, when the the fileIndex attribute is set to "max" but all the other settings are the same the following actions will be performed.

Number of rollovers	Active output target	Archived log files	Description
0	foo.log	-	All logging is going to the initial file.
1	foo.log	foo-1.log	During the first rollover foo.log is renamed to foo-1.log. A new foo.log file is created and starts being written to.
2	foo.log	foo-1.log, foo-2.log	During the second rollover foo.log is renamed to foo-2.log. A new foo.log file is created and starts being written to.
3	foo.log	foo-1.log, foo-2.log, foo-3.log	During the third rollover foo.log is renamed to foo-3.log. A new foo.log file is created and starts being written to.

4	foo.log	foo-1.log, foo-2.log, foo-3.log	In the fourth and subsequent rollovers, foo-1.log is deleted, foo-2.log is renamed to foo-1.log, foo-3.log is renamed to foo-2.log and foo.log is renamed to foo-3.log. A new foo.log file is created and starts being written to.

Parameter Name	Type	Description
fileIndex	String	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.
min	integer	The minimum value of the counter. The default value is 1.
max	integer	The maximum value of the counter. Once this values is reached older archives will be deleted on subsequent rollovers.
compressionLevel	integer	Sets the compression level, 0-9, where 0 = none, 1 = best speed, through 9 = best compression. Only implemented for ZIP files.

DefaultRolloverStrategy 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 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>
```

9.1.15 RollingRandomAccessFileAppender

As of beta-9, the name of this appender has been changed from FastRollingFile to RollingRandomAccessFile. Configurations using the `FastRollingFile` element no longer work and should be modified to use the `RollingRandomAccessFile` element.

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.

File locking is not supported by the RollingRandomAccessFileAppender.

Parameter Name	Type	Description
append	boolean	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.

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.
fileName	String	The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.
filePattern	String	The pattern of the file name of the archived log file. The format of the pattern should 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.
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
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.

RollingRandomAccessFileAppender Parameters

9.1.15.1 Triggering Policies

See RollingFileAppender Triggering Policies.

9.1.15.2 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 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.

```
<?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>
      <DefaultRolloverStrategy max="20"/>
    </RollingRandomAccessFile>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="RollingRandomAccessFile"/>
    </Root>
  </Loggers>
</Configuration>
```

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 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>
```

9.1.16 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.

Parameter Name	Type	Description
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.
name	String	The name of the Appender.
rewritePolicy	RewritePolicy	The RewritePolicy that will manipulate the LogEvent.
routes	Routes	Contains one or more Route declarations to identify the criteria for choosing Appenders.
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.

RoutingAppender Parameters

9.1.16.1 Routes

The Routes element accepts a single, required 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.

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.

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 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>
```

9.1.17 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.

Parameter Name	Type	Description
bcc	String	The comma-separated list of BCC email addresses.
cc	String	The comma-separated list of CC email addresses.

bufferSize	integer	The maximum number of log events to be buffered for inclusion in the message. Defaults to 512.
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.
from	String	The email address of the sender.
layout	Layout	The Layout to use to format the LogEvent. The default is SerializedLayout.
name	String	The name of the Appender.
replyTo	String	The comma-separated list of reply-to email addresses.
smtpDebug	boolean	When set to true enables session debugging on STDOUT. Defaults to false.
smtpHost	String	The SMTP hostname to send to. This parameter is required.
smtpPassword	String	The password required to authenticate against the SMTP server.
smtpPort	integer	The SMTP port to send to.
smtpProtocol	String	The SMTP transport protocol (such as "smtps", defaults to "smtp").
smtpUsername	String	The username required to authenticate against the SMTP server.
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.
to	String	The comma-separated list of recipient email addresses.

SMTPAppender Parameters

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <SMTP name="Mail" subject="Error Log" to="errors@logging.apache.org" from="test@logging.apache.org"
          smtpHost="localhost" smtpPort="25" bufferSize="50">
    </SMTP>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Mail"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.18 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. The default format is to send a Serialized LogEvent. Log4j 2 contains a SocketServer which is capable of receiving serialized LogEvents and routing them through the logging system on the server. You can optionally secure communication with SSL.

Parameter Name	Type	Description
name	String	The name of the Appender.
host	String	The name or address of the system that is listening for log events. This parameter is required.
port	integer	The port on the host that is listening for log events. This parameter must be specified.
protocol	String	"TCP" (default), "SSL" or "UDP".
SSL	SslConfiguration	Contains the configuration for the KeyStore and TrustStore.
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.
immediateFail	boolean	When set to true, log events will not wait to try to reconnect and will fail immediately if the socket is not available.
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.
layout	Layout	The Layout to use to format the LogEvent. The default is SerializedLayout.

reconnectionDelayMillis	integer	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`).
connectTimeoutMillis	integer	The connect timeout in milliseconds. The default is 0 (infinite timeout, like Socket.connect() methods).
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.

`SocketAppender` Parameters

This is an unsecured TCP configuration:

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

This is a secured SSL configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Socket name="socket" host="localhost" port="9500">
      <SerializedLayout />
      <SSL verifyHostName="true">
        <KeyStore location="log4j2-keystore.jks" password="changeme"/>
        <TrustStore location="truststore.jks" password="changeme"/>
      </SSL>
    </Socket>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="socket"/>
    </Root>
  </Loggers>
</Configuration>
```

9.1.19 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.

Parameter Name	Type	Description
advertise	boolean	Indicates whether the appender should be advertised.
appName	String	The value to use as the APP-NAME in the RFC 5424 syslog record.
charset	String	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.
connectTimeoutMillis	integer	The connect timeout in milliseconds. The default is 0 (infinite timeout, like Socket.connect() methods).
enterpriseNumber	integer	The IANA enterprise number as described in RFC 5424
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.

facility	String	The facility is used to try to classify the message. The facility option must be set to one of "KERN", "USER", "MAIL", "DAEMON", "AUTH", "SYSLOG", "LPR", "NEWS", "UUCP", "CRON", "AUTHPRIV", "FTP", "NTP", "AUDIT", "ALERT", "CLOCK", "LOCAL0", "LOCAL1", "LOCAL2", "LOCAL3", "LOCAL4", "LOCAL5", "LOCAL6", or "LOCAL7". These values may be specified as upper or lower case characters.
format	String	If set to "RFC5424" 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.
host	String	The name or address of the system that is listening for log events. This parameter is required.
id	String	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.
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.
immediateFail	boolean	When set to true, log events will not wait to try to reconnect and will fail immediately if the socket is not available.
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.

includeMDC	boolean	Indicates whether data from the ThreadContextMap will be included in the RFC 5424 Syslog record. Defaults to true.
loggerFields	List of KeyValuePairs	Allows arbitrary PatternLayout patterns to be included as specified ThreadContext fields; no default specified. To use, include a >LoggerFields< nested element, containing one or more >KeyValuePair< elements. Each >KeyValuePair< 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.
mdcExcludes	String	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.
mdcIncludes	String	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.
mdcRequired	String	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.
mdcPrefix	String	A string that should be prepended to each MDC key in order to distinguish it from event attributes. The default string is "mdc:". This attribute only applies to RFC 5424 syslog records.
messageId	String	The default value to be used in the MSGID field of RFC 5424 syslog records.
name	String	The name of the Appender.
newLine	boolean	If true, a newline will be appended to the end of the syslog record. The default is false.
port	integer	The port on the host that is listening for log events. This parameter must be specified.

protocol	String	"TCP" or "UDP". This parameter is required.
SSL	SslConfiguration	Contains the configuration for the KeyStore and TrustStore.
reconnectionDelayMillis	integer	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`).

`SyslogAppender` Parameters

A sample syslogAppender configuration that is configured with two `SyslogAppenders`, one using the BSD format and one using RFC 5424.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <Syslog name="bsd" host="localhost" port="514" protocol="TCP"/>
    <Syslog name="RFC5424" format="RFC5424" host="localhost" port="8514"
            protocol="TCP" appName="MyApp" includeMDC="true"
            facility="LOCAL0" enterpriseNumber="18060" newLine="true"
            messageId="Audit" id="App"/>
  </Appenders>
  <Loggers>
    <Logger name="com.mycorp" level="error">
      <AppenderRef ref="RFC5424"/>
    </Logger>
    <Root level="error">
      <AppenderRef ref="bsd"/>
    </Root>
  </Loggers>
</Configuration>
```

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.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <Appenders>
    <TLSSyslog name="bsd" host="localhost" port="6514">
      <SSL verifyHostName="true">
        <KeyStore location="log4j2-keystore.jks" password="changeme"/>
        <TrustStore location="truststore.jks" password="changeme"/>
      </SSL>
    </TLSSyslog>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="bsd"/>
    </Root>
  </Loggers>
</Configuration>
```

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.

10.1.1 JSONLayout

Appends a series of JSON events as strings serialized as bytes. This layout requires Jackson jar files (see pom.xml for details).

10.1.1.1 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.

A well-formed JSON document follows this pattern:

```
[
  {
    "logger":"com.foo.Bar",
    "timestamp":"1376681196470",
    "level":"INFO",
    "thread":"main",
    "message":"Message flushed with immediate flush=true"
  },
  {
    "logger":"com.foo.Bar",
    "timestamp":"1376681196471",
    "level":"ERROR",
    "thread":"main",
    "message":"Message flushed with immediate flush=true",
    "throwable":"java.lang.IllegalArgumentException: badarg\\n\\tat org.apache.logging.log4j.core.appender.JS
  }
]
```

If `complete="false"`, the appender does not write the JSON open array character "[" at the start of the document. and "]" and the end.

This approach enforces the independence of the JSONLayout and the appender where you embed it.

10.1.1.2 Encoding

Appenders using this layout should have their `charset` set to `UTF-8` or `UTF-16`, otherwise events containing non-ASCII characters could result in corrupted log files. The default charset is `UTF-8`.

10.1.1.3 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.

Parameter Name	Type	Description
charset	String	The character set to use when converting the HTML String to a byte array. The value must be a valid Charset. If not specified, UTF-8 will be used.
compact	boolean	If true, the appender does not use end-of-lines and indentation. Defaults to false.
eventEol	boolean	If true, the appender appends an end-of-line after each record. Defaults to false. Use with eventEol=true and compact=true to get one record per line.
complete	boolean	If true, the appender includes the JSON header and footer. Defaults to false.
properties	boolean	If true, the appender includes the thread context in the generated JSON. Defaults to false.
locationInfo	boolean	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.

JSON Layout Parameters

10.1.2 HTMLLayout

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

Parameter Name	Type	Description
charset	String	The character set to use when converting the HTML String to a byte array. The value must be a valid Charset. If not specified, the default system Charset will be used.

contentType	String	The value to assign to the Content-Type header. The default is "text/html".
locationInfo	boolean	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.
title	String	A String that will appear as the HTML title.

HTML Layout Parameters

10.1.3 PatternLayout

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.

Parameter Name	Type	Description
charset	String	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.
pattern	String	A composite pattern string of one or more conversion patterns from the table below.
replace	RegexReplacement	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.
alwaysWriteExceptions	boolean	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.
header	String	The optional header string to include at the top of each log file.
footer	String	The optional footer string to include at the bottom of each log file.
noConsoleNoAnsi	boolean	If `true` (default is false) and `System.console()` is null, do not output ANSI escape codes.

PatternLayout Parameters

Parameter Name	Type	Description
regex	String	A Java-compliant regular expression to match in the resulting string. See Pattern .
replacement	String	The string to replace any matched sub-strings with.

RegexReplacement Parameters

10.1.3.1 Patterns

The conversions that are provided with Log4j are:

Conversion Pattern	Description
c{precision} **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. If a precision specifier is given and it is an integer value, then only the corresponding number of right most components of the logger name will be printed. If the precision contains other non-integer characters then the name will be abbreviated based on the pattern. If the precision integer is less than one the right-most token will still be printed in full. By default the logger name is printed in full. **Conversion Pattern** / **Logger Name** / **Result** %c{1} / org.apache.commons.Foo / Foo %c{2} / org.apache.commons.Foo / commons.Foo %c{1.} / org.apache.commons.Foo / o.a.c.Foo %c{1.1.~.~} / org.apache.commons.test.Foo / o.a.~.~.Foo %c{.} / org.apache.commons.test.Foo / ....Foo
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}
date{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 formats are `DEFAULT`, `ABSOLUTE`, `COMPACT`, `DATE`, `ISO8601`, and `ISO8601_BASIC`.

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 ISO8601 format is assumed.

Pattern	Example
%d{DEFAULT}	2012-11-02 14:34:02,781
%d{ISO8601}	2012-11-02T14:34:02,781
%d{ISO8601_BASIC}	20121102T143402,781
%d{ABSOLUTE}	14:34:02,781
%d{DATE}	02 Nov 2012 14:34:02,781
%d{COMPACT}	20121102143402781
%d{HH:mm:ss,SSS}	14:34:02,781
%d{dd MMM yyyy HH:mm:ss,SSS}	02 Nov 2012 14:34:02,781
%d{HH:mm:ss}{GMT+0}	18:34:02
%d{UNIX}	1351866842
%d{UNIX_MILLIS}	1351866842781

%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 (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.

enc{pattern}
encode{pattern>

Escape newlines and HTML special characters in the specified pattern.

Allows HTML to be safely logged.

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. Anytime user provided data is logged, this can provide a safeguard.

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}}`

The replaced characters are:

Character	Replacement
'\r', '\n'	Removed from the pattern
&, <, >, ", ', /	Replaced with the corresponding HTML entity

ex| **exception**| **throwable**
{["none"
|"full"
|depth
|"short"
|"short.className"
|"short.fileName"
|"short.lineNumber"
|"short.methodName"
|"short.message"
|"short.localizedMessage"]}

Outputs the Throwable trace bound to the LoggingEvent, 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.

F
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.

The default colors for each level are:

Level	ANSI color
FATAL	Bright red
ERROR	Bright red
WARN	Yellow
INFO	Green
DEBUG	Cyan
TRACE	Black (looks dark grey)

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.

Inter Code	0	1	2	3	4	5	6	7
Norn	Black	Red	Gree	Yello	Blue	Mage	Cyan	White
Brigl	Black	Red	Gree	Yello	Blue	Mage	Cyan	White

Color table

You can use the default colors with:

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

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

```
%highlight{%d [%t] %-5level: %msg%n%throwable}
   {FATAL=white, ERROR=red, WARN=blue, INFO=black,
    DEBUG=green, TRACE=blue}
```

You can highlight only the a portion of the log event:

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

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

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

You can also use the STYLE key to use a predefined group of colors:

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

The STYLE value can be one of:

Style	Description
Default	See above
Logback	

Level	ANSI color
FATAL	Blinking bright red
ERROR	Bright red
WARN	Red
INFO	Blue
DEBUG	Normal
TRACE	Normal

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}}

l
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 the 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.

m **msg** **message**	Outputs the application supplied message associated with the logging event.
M **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 name of the marker, if one is present.
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.

p| **level**{ *level*= *label*, *level*= *label*, ...} **p**| **level**{length= *n*} **p**| **level**{lowerCase= *true*| *false*}	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:
	`%level{WARN=Warning, DEBUG=Debug, ERROR=Error, TRACE=Trace, IN`
	
	Alternatively, for the compact-minded:
	`%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:
	`%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:
	`%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):
	`%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.
rEx["none"|"short"|"full"|depth],[filters(packages)} **rException**["none"|"short"|"full"|depth],[filters(packages)} **rThrowable**["none"|"short"|"full"|depth],[filters(packages)}	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 stacktrace will be printed. The conversion word can also be followed by "filters(packages)" where packages is a list of package names that should be suppressed from stack traces. Specifying **%rEx{none}** or **%rEx{0}** will suppress printing of the exception.
sn **sequenceNumber**	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.

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

For example:

```
%style{%d{ISO8601}}{black} %style{[%t]}{blue} %style{%-5level
```

You can also combine styles:

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

t **thread**	Outputs the name of the thread that generated the logging event.
x **NDC**	Outputs the Thread Context Stack (also known as the Nested Diagnostic Context or NDC) associated with the thread that generated the logging event.
X{key} **mdc**{key} **MDC**{key}	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 the key for the map placed between braces, as in **%X{clientNumber}** where `clientNumber` is the key. The value in the MDC corresponding to the key will be output. If no additional sub-option is specified, then the entire contents of the MDC key value pair set is output using a format {{key1,val1},{key2,val2}} See the ThreadContext class for more details.
u{"RANDOM" | "TIME"} **uuid**	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.
xEx{"none"|"short"|"full"|depth],[filters(packages)} **xException**["none"|"short"|"full"|depth],[filters(packages)} **xThrowable**["none"|"short"|"full"|depth],[filters(packages)}	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 stacktrace will be printed. The conversion word can also be followed by "filters(packages)" where packages is a list of package names that should be suppressed from stack traces. Specifying **%xEx{none}** or **%xEx{0}** will suppress printing of the exception.
%	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, it 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.

Format modifier	left justify	minimum width	maximum width	comment
%20c	false	20	none	Left pad with spaces if the category name is less than 20 characters long.
%-20c	true	20	none	Right pad with spaces if the category name is less than 20 characters long.
%.30c	NA	none	30	Truncate from the beginning if the category name is longer than 30 characters.
%20.30c	false	20	30	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.

%-20.30c	true	20	30	Right 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.
%-20.-30c	true	20	30	Right 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 end.

Pattern Converters

10.1.3.2 ANSI Styling on Windows

ANSI escape sequences are supported natively on many platforms but are not by default on Windows. To enable ANSI support simply add the Jansi jar to your application and Log4j will automatically make use of it when writing to the console.

10.1.3.3 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>
...
<PatternLayout pattern="%m%xEx{filters(${filters})}%n"/>
```

The result printed to the console will appear similar to:

```
Exception java.lang.IllegalArgumentException: IllegalArgument
         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>
```

10.1.4 RFC5424Layout

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.

Parameter Name	Type	Description
appName	String	The value to use as the APP-NAME in the RFC 5424 syslog record.
charset	String	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.
enterpriseNumber	integer	The IANA enterprise number as described in RFC 5424
exceptionPattern	String	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.
facility	String	The facility is used to try to classify the message. The facility option must be set to one of "KERN", "USER", "MAIL", "DAEMON", "AUTH", "SYSLOG", "LPR", "NEWS", "UUCP", "CRON", "AUTHPRIV", "FTP", "NTP", "AUDIT", "ALERT", "CLOCK", "LOCAL0", "LOCAL1", "LOCAL2", "LOCAL3", "LOCAL4", "LOCAL5", "LOCAL6", or "LOCAL7". These values may be specified as upper or lower case characters.

format	String	If set to "RFC5424" 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.
id	String	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.
immediateFlush	boolean	When set to true, each write will be followed by a flush. This will guarantee the data is written to disk but could impact performance.
includeMDC	boolean	Indicates whether data from the ThreadContextMap will be included in the RFC 5424 Syslog record. Defaults to true.
loggerFields	List of KeyValuePairs	Allows arbitrary PatternLayout patterns to be included as specified ThreadContext fields; no default specified. To use, include a <LoggerFields> nested element, containing one or more <KeyValuePair> elements. Each <KeyValuePair> 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.
mdcExcludes	String	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.
mdcIncludes	String	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.

mdcRequired	String	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.
mdcPrefix	String	A string that should be prepended to each MDC key in order to distinguish it from event attributes. The default string is "mdc:". This attribute only applies to RFC 5424 syslog records.
mdcId	String	A required MDC ID. This attribute only applies to RFC 5424 syslog records.
messageId	String	The default value to be used in the MSGID field of RFC 5424 syslog records.
newLine	boolean	If true, a newline will be appended to the end of the syslog record. The default is false.
newLineEscape	String	String that should be used to replace newlines within the message text.

RFC5424Layout Parameters

10.1.5 SerializedLayout

The SerializedLayout simply serializes the LogEvent into a byte array. This is useful when sending messages via JMS or via a Socket connection. The SerializedLayout accepts no parameters.

10.1.6 SyslogLayout

The SyslogLayout formats the LogEvent as BSD Syslog records matching the same format used by Log4j 1.2.

Parameter Name	Type	Description
charset	String	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.

facility	String	The facility is used to try to classify the message. The facility option must be set to one of "KERN", "USER", "MAIL", "DAEMON", "AUTH", "SYSLOG", "LPR", "NEWS", "UUCP", "CRON", "AUTHPRIV", "FTP", "NTP", "AUDIT", "ALERT", "CLOCK", "LOCAL0", "LOCAL1", "LOCAL2", "LOCAL3", "LOCAL4", "LOCAL5", "LOCAL6", or "LOCAL7". These values may be specified as upper or lower case characters.
newLine	boolean	If true, a newline will be appended to the end of the syslog record. The default is false.
newLineEscape	String	String that should be used to replace newlines within the message text.

SyslogLayout Parameters

10.1.7 XMLLayout

Appends a series of `Event` elements as defined in the log4j.dtd.

10.1.7.1 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<Events xmlns="http://logging.apache.org/log4j/2.0/events">
  <Event logger="com.foo.Bar" timestamp="1373436580419" level="INFO" thread="main">
    <Message><![CDATA[This is a log message 1]]></Message>
    <Marker parent="Parent Marker"><Child Marker></Marker>
  </Event>
  <Event logger="com.foo.Baz" timestamp="1373436580420" level="INFO" thread="main">
    <Message><![CDATA[This is a log message 2]]></Message>
    <Marker><The Marker Name></Marker>
  </Event>
</Events>
```

If `complete="false"`, the appender does not write the XML processing instruction and the root element.

This approach enforces the independence of the XMLLayout and the appender where you embed it.

10.1.7.2 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. Only the leaf marker is included, not the full hierarchy.

10.1.7.3 Encoding

Appenders using this layout should have their `charset` set to `UTF-8` or `UTF-16`, otherwise events containing non ASCII characters could result in corrupted log files.

10.1.7.4 Pretty vs. compact XML

By default, the XML 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 XML. If `compact="true"`, then no end-of-line or indentation is used. Message content may contain, of course, end-of-lines.

10.1.8 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 Graylog2 server:

```
[
  <Appenders>
    <Socket name="Graylog" protocol="udp" host="graylog.domain.com" port="12201">
        <GelfLayout host="someserver" compressionType="GZIP" compressionThreshold="1024">
            <KeyValuePair key="additionalField1" value="additional value 1"/>
            <KeyValuePair key="additionalField2" value="additional value 2"/>
        </GelfLayout>
    </Socket>
  </Appenders>
]
```

See also:

- The GELF home page
- The GELF specification

10.1.9 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 4 - 20 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.

Parameter Name	Type	Description
level	String	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.
rate	float	The average number of events per second to allow.
maxBurst	integer	The maximum number of events that can occur before events are filtered for exceeding the average rate. The default is 10 times the rate.
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.

Burst Filter Parameters

A configuration containing the BurstFilter might look like:

```
<?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"/>
    </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>
    <Marker 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.

Parameter Name	Type	Description
defaultThreshold	String	Level of messages to be filtered. If there is no matching key in the key/value pairs then this level will be compared against the event's level.
keyValuePair	KeyValuePair[]	One or more KeyValuePair elements that define the matching value for the key and the Level to evaluate when the key matches.

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.

Dynamic Threshold Filter Parameters

Here is a sample configuration containing the DynamicThresholdFilter:

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

Parameter Name	Type	Description
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.

Map Filter Parameters

As in this configuration, the MapFilter can be used to log particular events:

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

```
<?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">
      <MapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
        <KeyValuePair key="eventId" value="Login"/>
        <KeyValuePair key="eventId" value="Logout"/>
      </MapFilter>
      <AppenderRef ref="RollingFile">
      </AppenderRef>
    </Root>
  </Loggers>
</Configuration>
```

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.

```
<?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">
        <MapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
          <KeyValuePair key="eventId" value="Login"/>
          <KeyValuePair key="eventId" value="Logout"/>
        </MapFilter>
      </AppenderRef>
    </Root>
  </Loggers>
</Configuration>
```

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.

Parameter Name	Type	Description
marker	String	The name of the Marker to compare.
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.

Marker Filter Parameters

A sample configuration that only allows the event to be written by the appender if the Marker matches:

```
<?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>
      <TimeBasedTriggeringPolicy />
    </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.

Parameter Name	Type	Description
regex	String	The regular expression.

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 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">
      <RegexFilter 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 StructuredDataFilter

The StructuredDataFilter is a MapFilter that also allows filtering on the event id, type and message.

Parameter Name	Type	Description

keyValuePair	KeyValuePair[]	One or more KeyValuePair elements that define the key in the map and the value to match on. "id", "id.name", "type", and "message" 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 "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.

StructuredData Filter Parameters

As in this configuration, the StructuredDataFilter can be used to log particular events:

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

11.1.8 ThreadContextMapFilter

The ThreadContextMapFilter allows filtering against data elements that are in the ThreadContext Map.

Parameter Name	Type	Description
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.

ThreadContext Map Filter Parameters

A configuration containing the ThreadContextMapFilter might look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="MyApp" packages="">
  <ThreadContextMapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
    <KeyValuePair key="User1" value="DEBUG"/>
    <KeyValuePair key="User2" value="WARN"/>
  </ThreadContextMapFilter>
  <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 ThreadContextMapFilter 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"/>
      <ThreadContextMapFilter onMatch="ACCEPT" onMismatch="NEUTRAL" operator="or">
        <KeyValuePair key="foo" value="bar"/>
        <KeyValuePair key="User2" value="WARN"/>
      </ThreadContextMapFilter>
    </Root>
  </Loggers>
</Configuration>
```

11.1.9 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.

Parameter Name	Type	Description
level	String	A valid Level name to match on.
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.

Threshold Filter Parameters

A sample configuration that only allows the event to be written by the appender if the level matches:

```
<?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">
      <ThresholdFilter level="TRACE" 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>
```

11.1.10 TimeFilter

The time filter can be used to restrict filter to only a certain portion of the day.

Parameter Name	Type	Description
start	String	A time in HH:mm:ss format.
end	String	A time in HH:mm:ss format. Specifying an end time less than the start time will result in no log entries being written.
timezone	String	The timezone to use when comparing to the event timestamp.
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.

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:

```
<?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">
      <TimeFilter start="05:00:00" end="05:30:00" 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>
```

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 to 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.
- **Asynchronous Appenders** already existed in Log4j 1.x, but 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.)
- (For synchronous and asynchronous use) **Random Access File Appenders** are an alternative to Buffered File Appenders. Under the hood, these new appenders use a ByteBuffer + RandomAccessFile instead of a BufferedOutputStream. In our testing this was about 20-200% faster. These appenders can also be used with synchronous loggers and will give the same performance benefits. Random Access File Appenders 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 throughput. With an asynchronous logger your application can log messages at 6 - 68 times the rate of a synchronous logger.
- Lower logging latency. Latency is the time it takes for a call to Logger.log to return. Asynchronous Loggers have consistently lower latency than synchronous loggers or even queue-based asynchronous appenders. Applications interested in low latency often care not only about average latency, but also about worst-case latency. Our performance comparison shows that Asynchronous Loggers also do better when comparing the maximum latency of 99% or even 99.99% of observations with other logging methods.
- Prevent or dampen latency spikes during bursts of events. If the queue size is configured large enough to handle spikes, asynchronous logging will help prevent your application from falling behind (as much) during sudden bursts of activity.

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.

12.1.2 Making All Loggers Asynchronous

Requires disruptor-3.0.0.jar or higher on the classpath.

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 `Log4jContextSelector` 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 version="1.0" encoding="UTF-8"?>

<!-- Don't forget to set system property
-DLog4jContextSelector=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.

System Property	Default Value	Description
AsyncLogger.ExceptionHandler	`null`	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.
AsyncLogger.RingBufferSize	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.

AsyncLogger.WaitStrategy	`Sleep`	Valid values: Block, 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. `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.
AsyncLogger.ThreadNameStrategy	`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.

log4j.Clock	`SystemClock`	Implementation of the `org.apache.logging.log4j.core.helpers` 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 a fully qualified class name of a custom class that implements the `Clock` interface.

System Properties to configure all asynchronous loggers

12.1.3 Mixing Synchronous and Asynchronous Loggers

Requires disruptor-3.0.0.jar or higher on the classpath. 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. The same configuration file 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 version="1.0" encoding="UTF-8"?>

<!-- No need to set system property "Log4jContextSelector" 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 %p %class{1.} [%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.

System Property	Default Value	Description
AsyncLoggerConfig.ExceptionHandle	`null`	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.
AsyncLoggerConfig.RingBufferSize	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.

AsyncLoggerConfig.WaitStrategy	`Sleep`	Valid values: Block, 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. `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

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 4 - 20 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 performance results below were all derived from running the PerfTest, MTPerfTest and PerfTestDriver classes which can be found in the Log4j 2 unit test source directory. All tests were done using the default settings (SystemClock and SleepingWaitStrategy). The methodology used was the same for all tests:

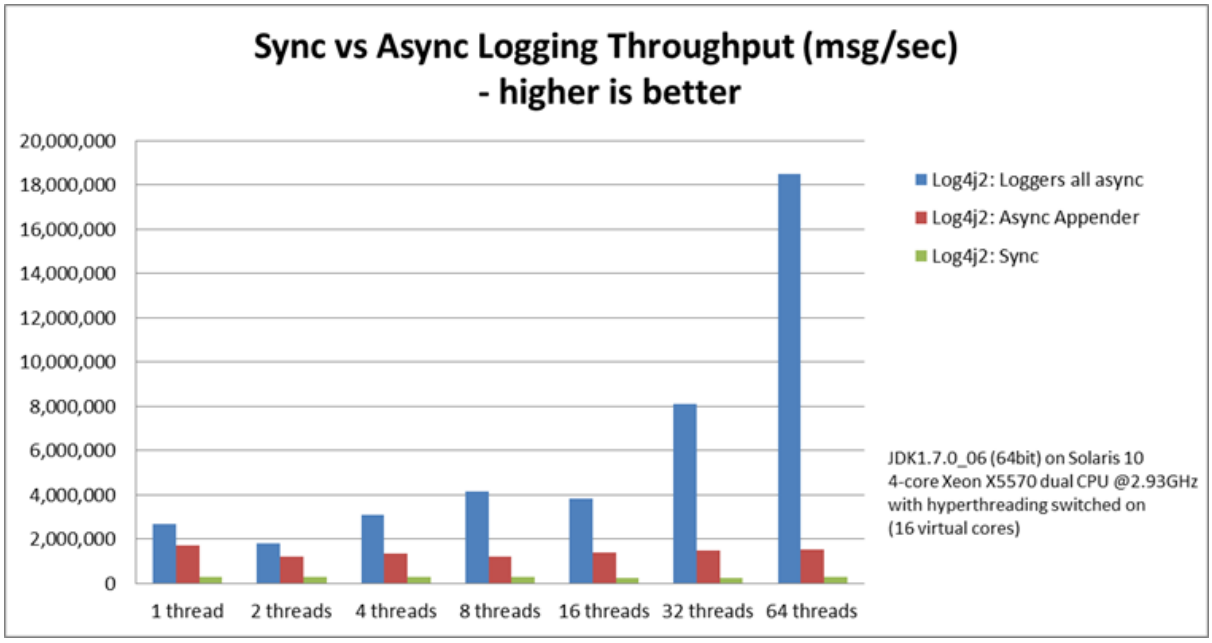
- First, warm up the JVM by logging 200,000 log messages of 500 characters.
- Repeat the warm-up 10 times, then wait 10 seconds for the I/O thread to catch up and buffers to drain.
- Latency test: at less than saturation, measure how long a call to Logger.log takes. Pause for 10 microseconds * threadCount between measurements. Repeat this 5 million times, and measure average latency, latency of 99% of observations and 99.99% of observations.
- Throughput test: 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 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.

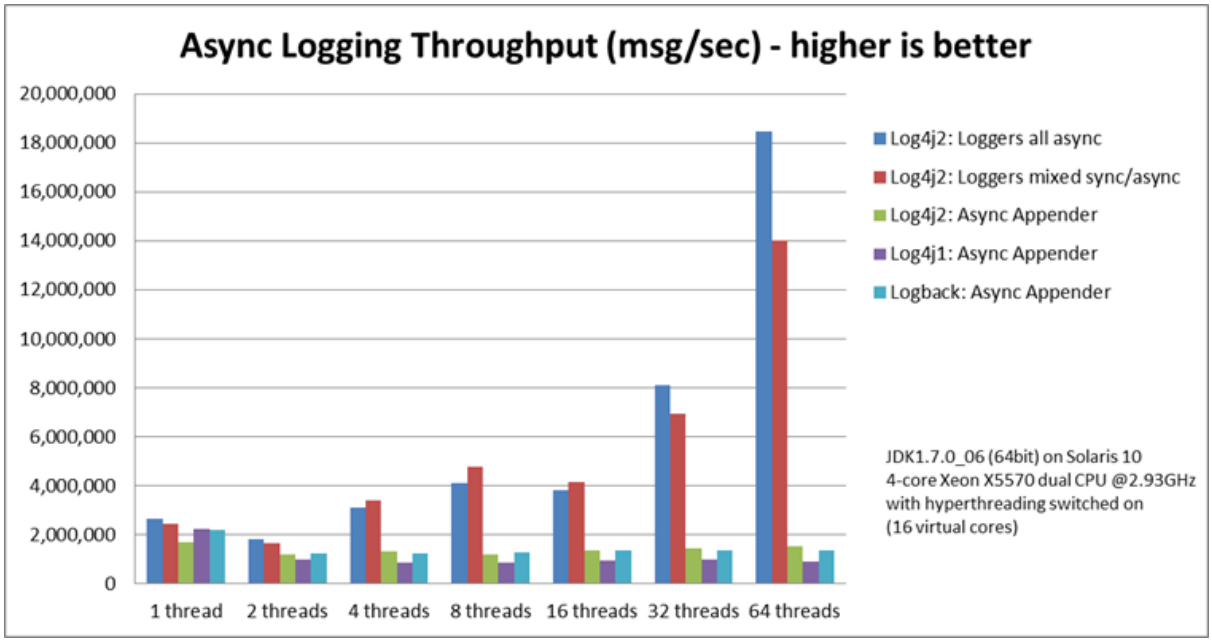
Asynchronous loggers' throughput increases with the number of threads, whereas both synchronous loggers and asynchronous appenders have more or less constant throughput regardless of the number of threads that are doing the logging.



12.1.5.2 Asynchronous Throughput Comparison with Other Logging Packages

We also compared 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):

Logger	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
Log4j 2: Loggers all asynchronous	2,652,412	909,119	776,993	516,365	239,246	253,791	288,997
Log4j 2: Loggers mixed sync/ async	2,454,358	839,394	854,578	597,913	261,003	216,863	218,937
Log4j 2: Async Appender	1,713,429	603,019	331,506	149,408	86,107	45,529	23,980
Log4j1: Async Appender	2,239,664	494,470	221,402	109,314	60,580	31,706	14,072
Logback: Async Appender	2,206,907	624,082	307,500	160,096	85,701	43,422	21,303
Log4j 2: Synchronous	273,536	136,523	67,609	34,404	15,373	7,903	4,253
Log4j1: Synchronous	326,894	105,591	57,036	30,511	13,900	7,094	3,509
Logback: Synchronous	178,063	65,000	34,372	16,903	8,334	3,985	1,967

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):

Logger	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Log4j 2: Loggers all asynchronous	1,715,344	928,951	1,045,265	1,509,109	1,708,989	773,565
Log4j 2: Loggers mixed sync/ async	571,099	1,204,774	1,632,204	1,368,041	462,093	908,529
Log4j 2: Async Appender	1,236,548	1,006,287	511,571	302,230	160,094	60,152
Log4j1: Async Appender	1,373,195	911,657	636,899	406,405	202,777	162,964
Logback: Async Appender	1,979,515	783,722	582,935	289,905	172,463	133,435
Log4j 2: Synchronous	281,250	225,731	129,015	66,590	34,401	17,347
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 Throughput of Logging With Location (includeLocation="true")

On Solaris 10 (64bit) with JDK1.7.0_06, 4-core Xeon X5570 dual CPU @2.93Ghz with hyperthreading switched off (8 virtual cores):

Logger (Log4j 2)	1 thread	2 threads	4 threads	8 threads
Loggers all asynchronous	75,862	88,775	80,240	68,077
Loggers mixed sync/ async	61,993	66,164	55,735	52,843
Async Appender	47,033	52,426	50,882	36,905
Synchronous	31,054	33,175	29,791	23,628

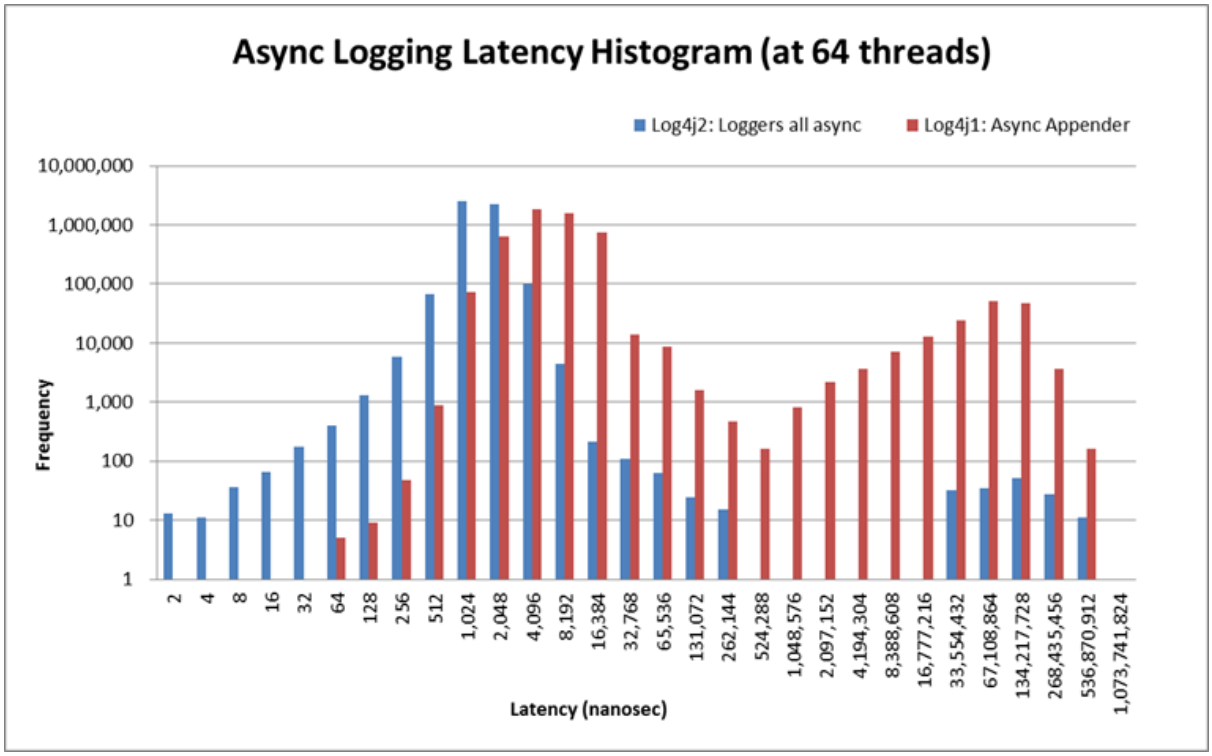
Throughput in log messages/second per thread

As expected, logging location information has a large performance impact. Asynchronous loggers are 4 - 20 times slower, while synchronous loggers are 1.3 - 5 times slower. However, if you do need location information, asynchronous logging will still be faster than synchronous logging.

12.1.5.4 Latency

Latency tests are done by logging at less than saturation, measuring how long a call to Logger.log takes to return. After each call to Logger.log, the test waits for 10 microseconds * threadCount before continuing. Each thread logs 5 million messages.

All the latency measurements below are results of tests run on Solaris 10 (64bit) with JDK1.7.0_06, 4-core Xeon X5570 dual CPU @2.93Ghz with hyperthreading switched on (16 virtual cores).



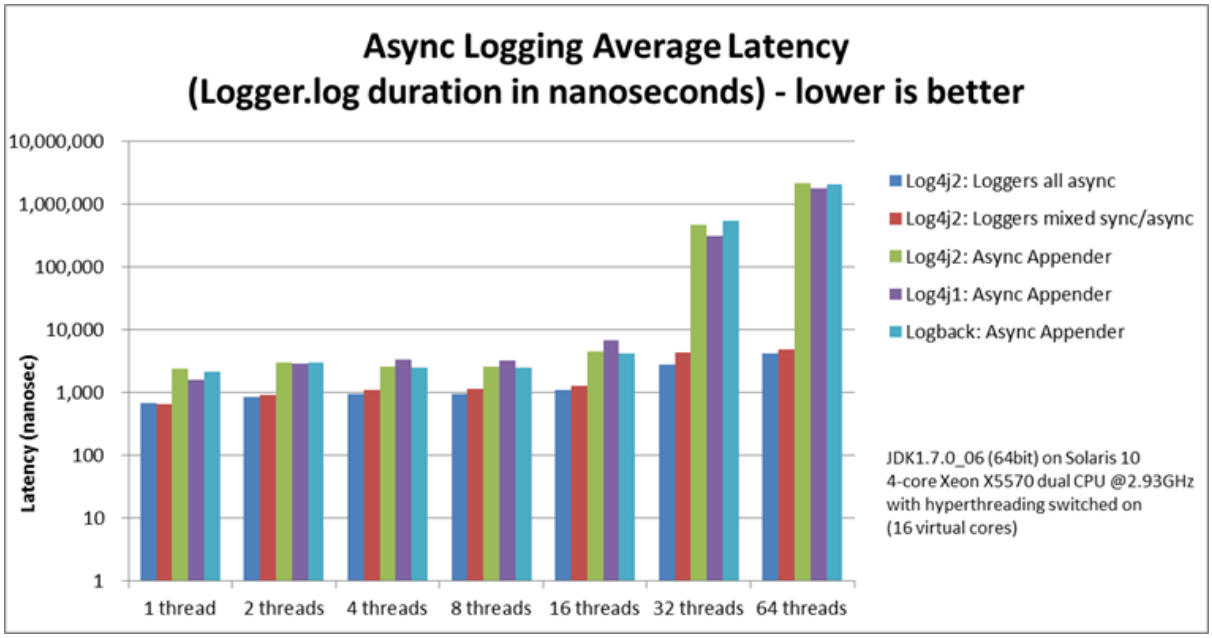
Note that this is log-scale, not linear. The above graph compares the latency distributions of an asynchronous logger and a Log4j 1.2.17 Async Appender. This shows the latency of one thread during a test where 64 threads are logging in parallel. The test was run once for the async logger and once for the async appender.

	Average latency		99% observations less than		99.99% observations less than	
	1 thread	64 threads	1 thread	64 threads	1 thread	64 threads
Log4j 2: Loggers all async	677	4,135	1,638	4,096	8,192	16,128
Log4j 2: Loggers mixed sync/ async	648	4,873	1,228	4,096	8,192	16,384
Log4j 2: Async Appender	2,423	2,117,722	4,096	67,108,864	16,384	268,435,456

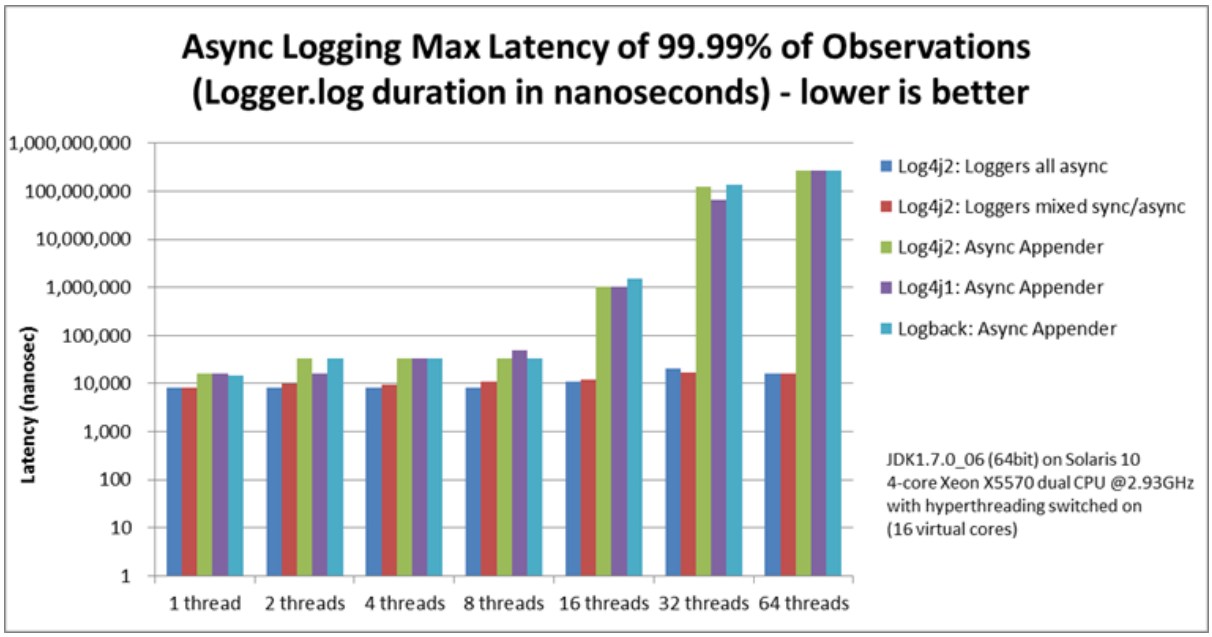
Log4j1: Async Appender	1,562	1,781,404	4,096	109,051,904	16,384	268,435,456
Logback: Async Appender	2,123	2,079,020	3,276	67,108,864	14,745	268,435,456

Latency of a call to Logger.log() in nanoseconds

The latency comparison graph below is also log-scale, and shows the average latency of asynchronous loggers and ArrayBlockingQueue-based asynchronous appenders in scenarios with more and more threads running in parallel. Up to 8 threads asynchronous appenders have comparable average latency, two or three times that of asynchronous loggers. With more threads, the average latency of asynchronous appenders is orders of magnitude larger than asynchronous loggers.



Applications interested in low latency often care not only about average latency, but also about worst-case latency. The graph below shows that asynchronous loggers also do better when comparing the maximum latency of 99.99% of observations with other logging methods. When increasing the number of threads the vast majority of latency measurements for asynchronous loggers stay in the 10-20 microseconds range where Asynchronous Appenders start experiencing many latency spikes in the 100 millisecond range, a difference of four orders of magnitude.



12.1.5.5 FileAppender vs. RandomAccessFileAppender

The appender comparison below was done with *synchronous loggers*.

On Windows 7 (64bit) with JDK1.7.0_11, 2-core Intel i5-3317u CPU @1.70Ghz with hyperthreading switched on (4 virtual cores):

Appender	1 thread	2 threads	4 threads	8 threads
RandomAccessFileAp	250,438	169,939	109,074	58,845
FileAppender	186,695	118,587	57,012	28,846
RollingRandomAccess	278,369	213,176	125,300	63,103
RollingFileAppender	182,518	114,690	55,147	28,153

Throughput per thread in messages/second

On Solaris 10 (64bit) with JDK1.7.0_06, 4-core dual Xeon X5570 CPU @2.93GHz with hyperthreading switched off (8 virtual cores):

Appender	1 thread	2 threads	4 threads	8 threads
RandomAccessFileAp	240,760	128,713	66,555	30,544
FileAppender	172,517	106,587	55,885	25,675
RollingRandomAccess	228,491	135,355	69,277	32,484
RollingFileAppender	186,422	97,737	55,766	25,097

Throughput per thread in messages/second

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 JMX

13.1 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.

13.2 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.disable.jmx=true` when you start the Java VM.

13.2.1 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.

13.2.2 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
```

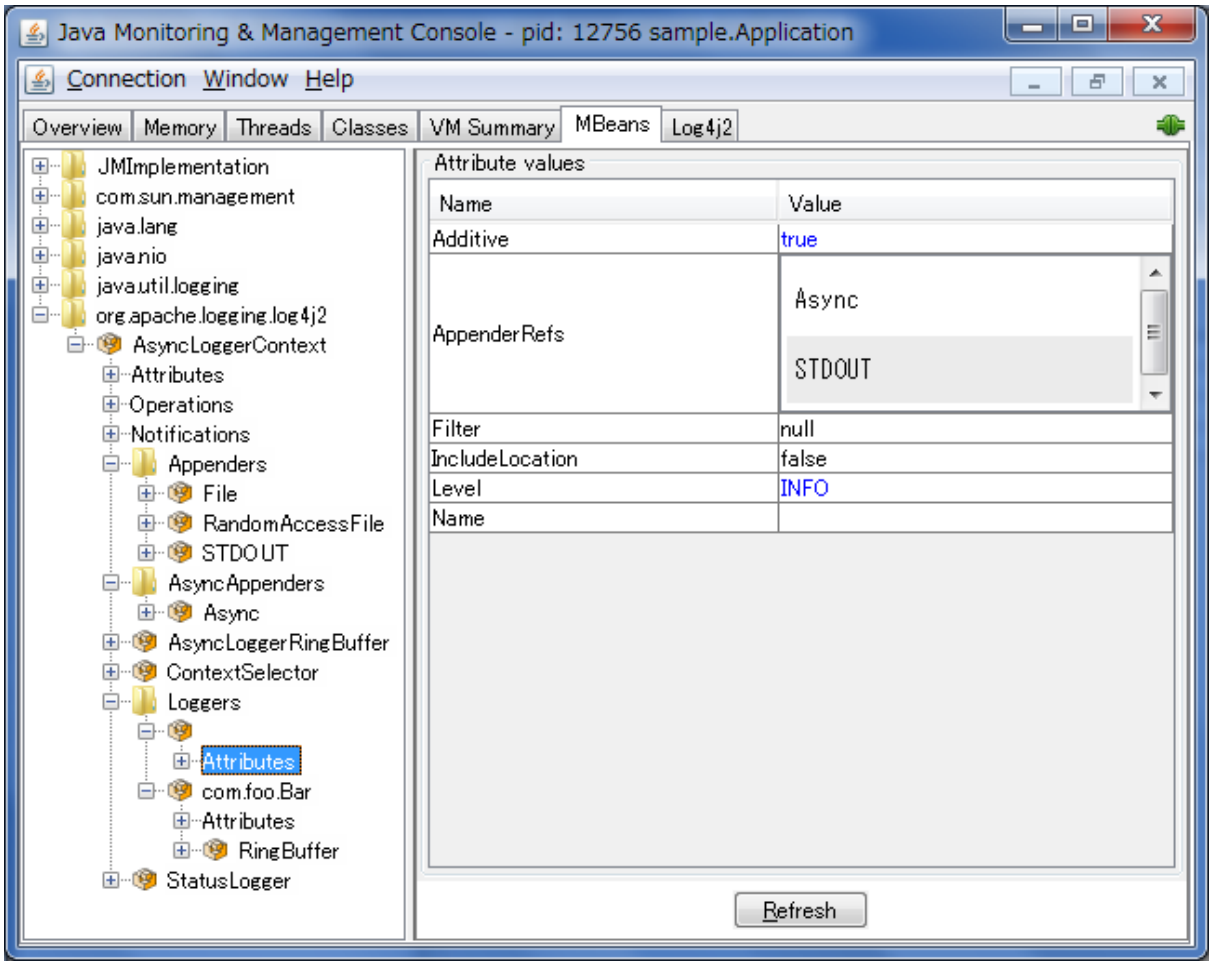
In the property above, `portNum` is the port number through which you want to enable JMX RMI connections.

For more details, see Oracle's documentation on Remote Monitoring and Management.

13.3 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 or by exploring directly in JConsole.

The screenshot below shows the Log4j MBeans in JConsole.



13.4 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.

13.4.1 Running the Client GUI as a JConsole Plug-in

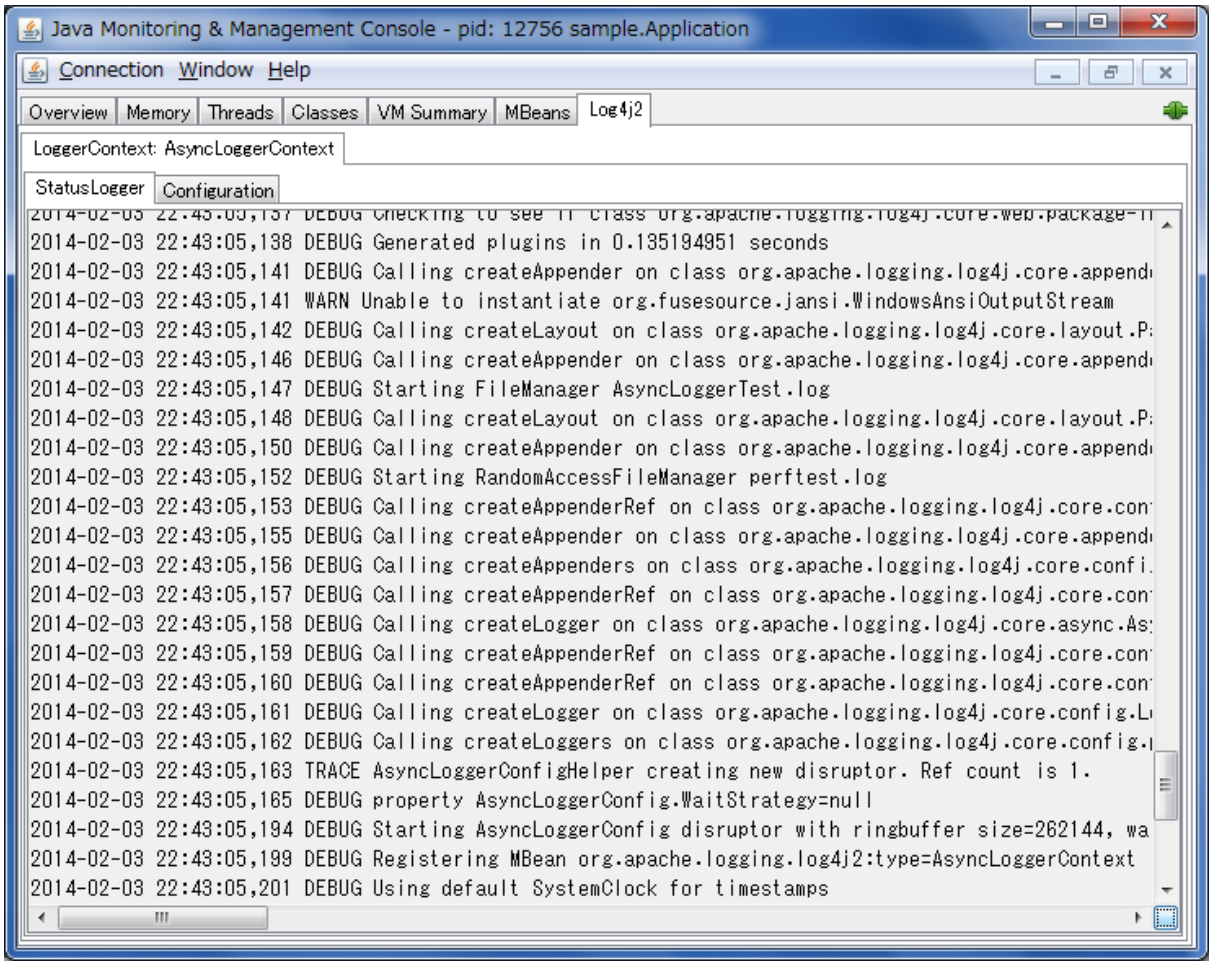
To run the Log4j JMX Client GUI as a JConsole Plug-in, start JConsole with the following command:

```
$JAVA_HOME/bin/jconsole -pluginpath /path/to/log4j-api-
${Log4jReleaseVersion}.jar:/path/to/log4j-core-${Log4jReleaseVersion}.jar:/
path/to/log4j-jmx-gui-${Log4jReleaseVersion}.jar
```

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
```

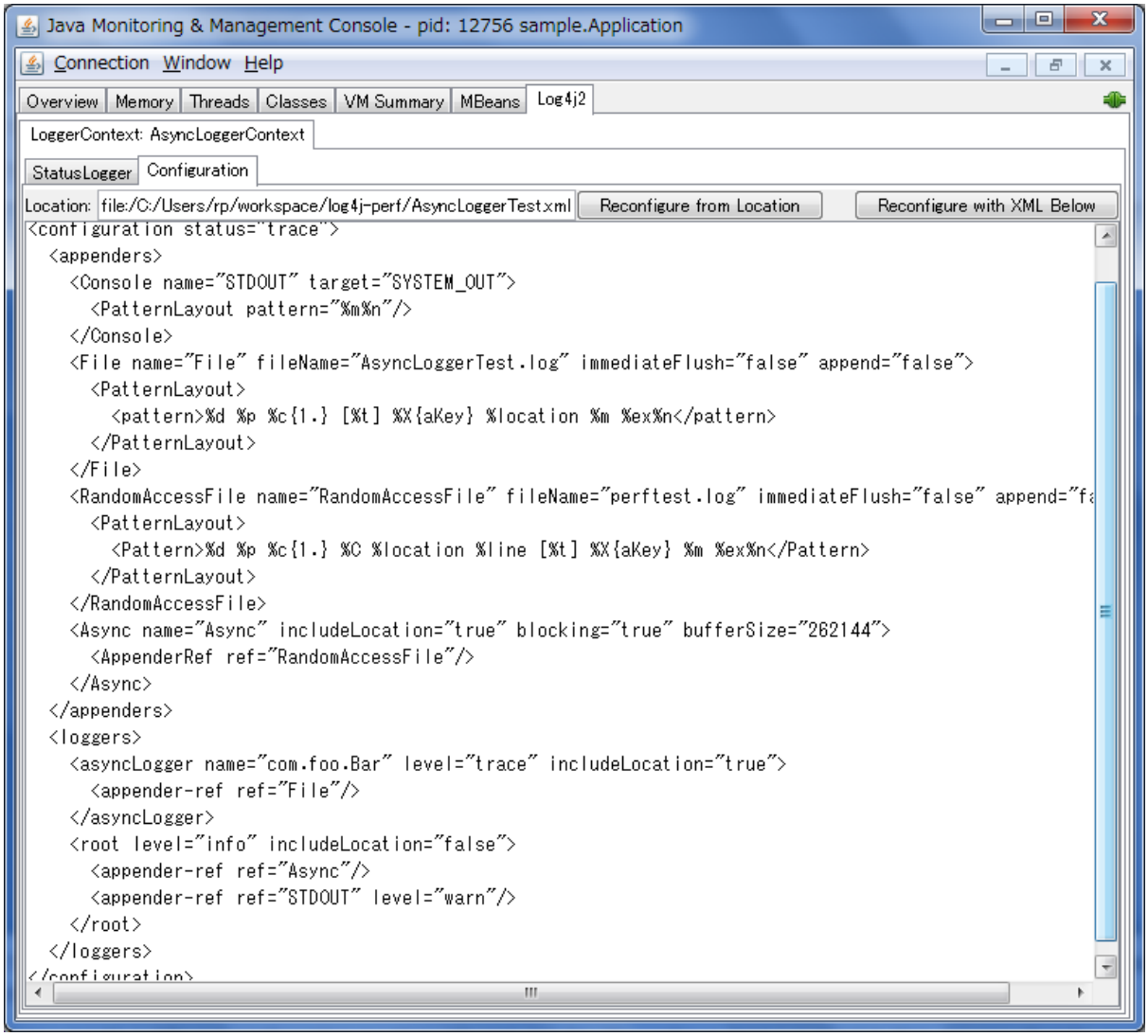
If you execute the above command and connect to your application, you will see an extra "Log4j 2" tab in the JConsole window. This tab contains the client GUI, with the StatusLogger selected. The screenshot below shows the StatusLogger panel in JConsole.



13.4.2 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.

13.4.3 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:

```
$JAVA_HOME/bin/java -cp /path/to/log4j-api-${Log4jReleaseVersion}.jar:/
path/to/log4j-core-${Log4jReleaseVersion}.jar:/path/to/log4j-jmx-gui-
${Log4jReleaseVersion}.jar org.apache.logging.log4j.jmx.gui.ClientGui
<options>
```

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.)

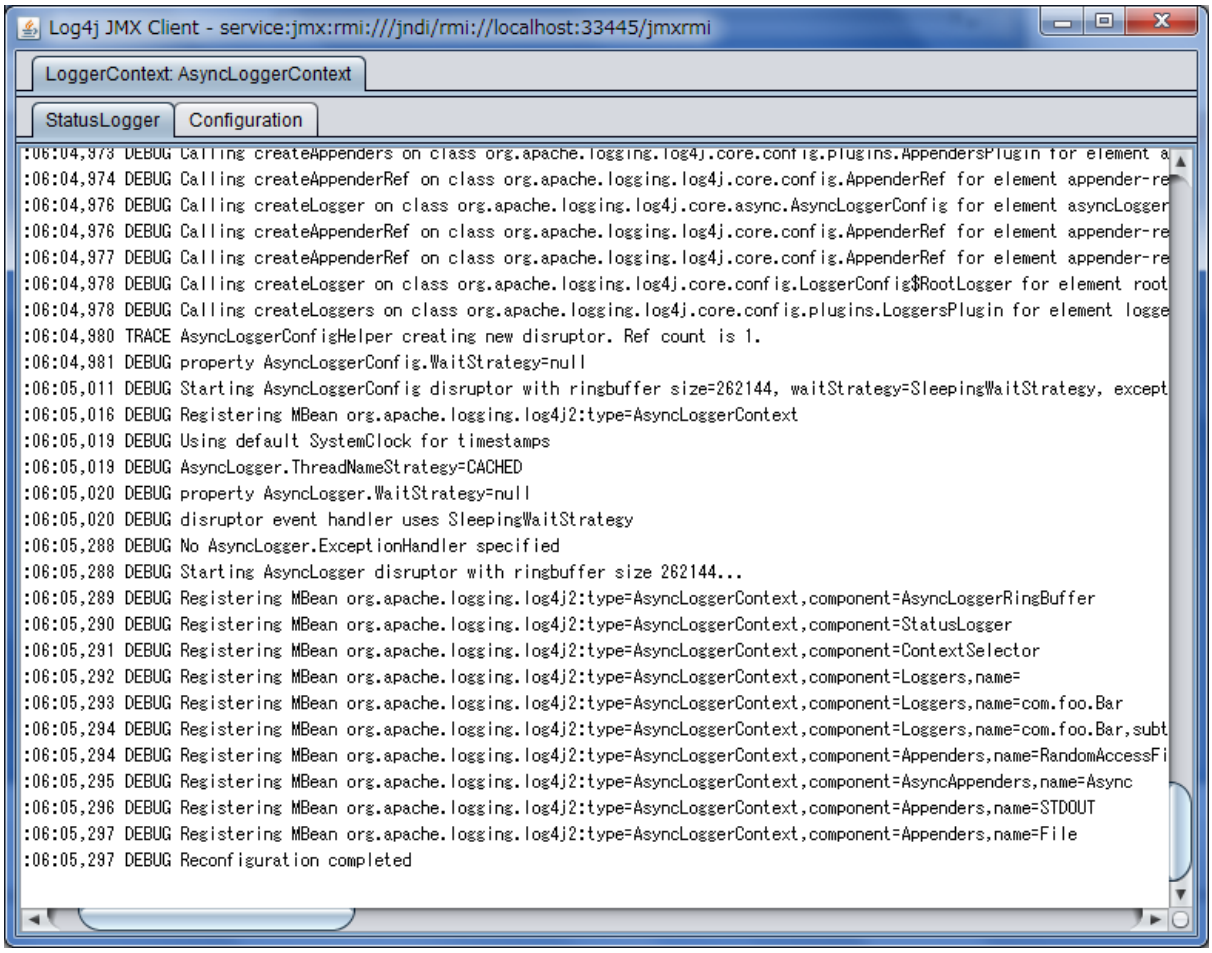
Then you can run the client with this command:

```
$JAVA_HOME/bin/java -cp /path/to/log4j-api-${Log4jReleaseVersion}.jar:/
path/to/log4j-core-${Log4jReleaseVersion}.jar:/path/to/log4j-jmx-gui-
${Log4jReleaseVersion}.jar org.apache.logging.log4j.jmx.gui.ClientGui
localhost:33445
```

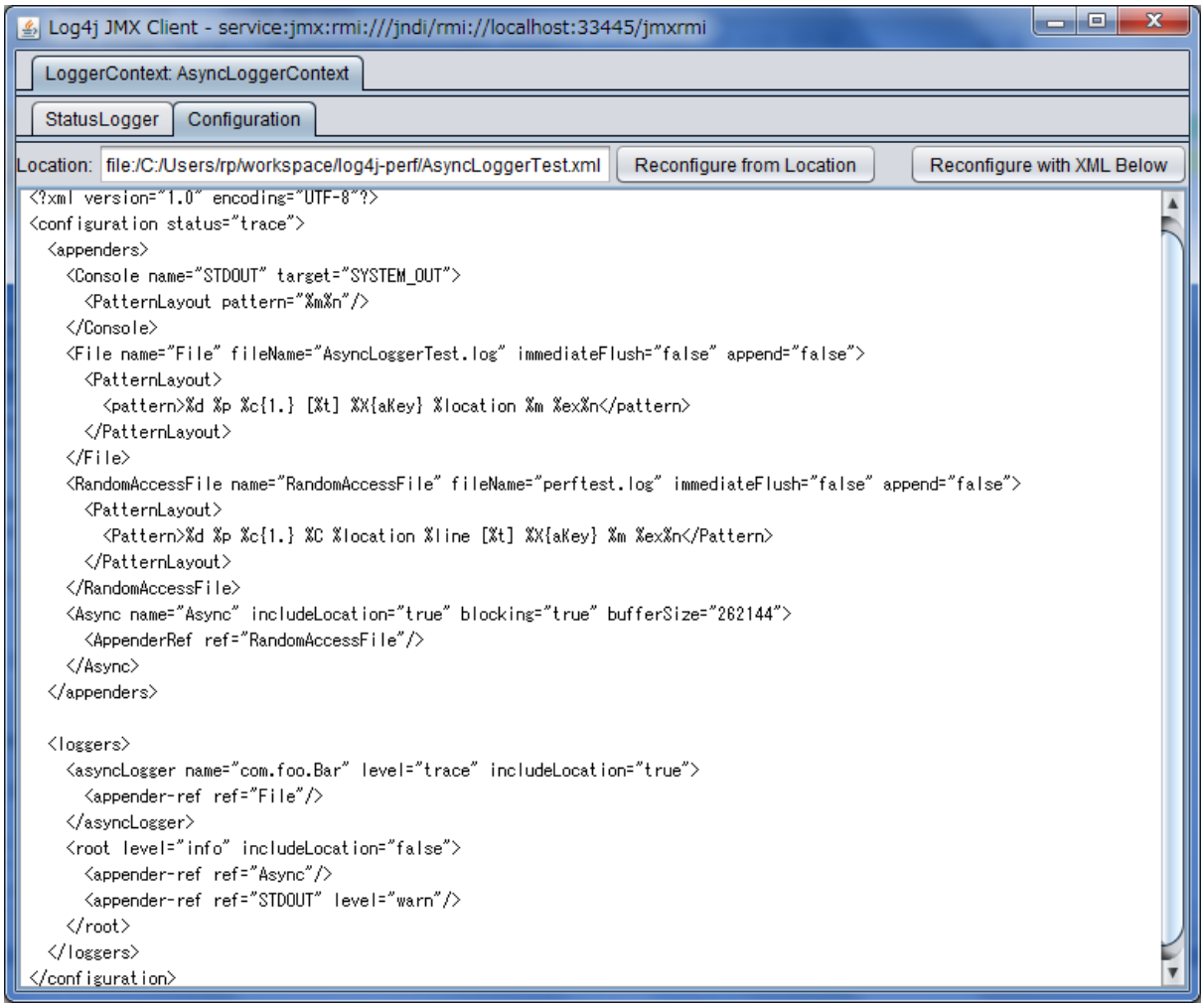
or on Windows:

```
%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.



14 Logging Separation

14.1 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.

14.1.1 Use Cases

This section describes some of the use cases where Log4j could be used and what its desired behavior might be.

14.1.1.1 Standalone Application

Standalone applications are usually relatively simple. They typically have one bundled executable that requires only a single logging configuration.

14.1.1.2 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.

14.1.1.3 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.

14.1.1.4 "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.

14.1.1.5 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`.

14.1.2 Approaches

14.1.2.1 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.

14.1.2.2 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 "Log4jContextSelector" 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 `Log4jContextSelector` 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. Note that the JndiContextSelector will not work unless `log4j2.enableJndiContextSelector=true` is set as a system property or environment variable. See the enableJndiContextSelector system property.

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.

15 Extending Log4j

15.1 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.

15.1.1 LoggerContextFactory

The `LoggerContextFactory` binds the Log4j API to its implementation. The Log4j `LogManager` locates a `LoggerContextFactory` by locating all instances of `META-INF/log4j-provider.properties`, a standard `java.util.Properties` file, and then inspecting each to verify that it specifies a value for the Log4jAPIVersion property that conforms to the version required by the `LogManager`. If more than one valid implementation is located the value for FactoryPriority will be used to identify the factory with the highest priority. Finally, the value of the LoggerContextFactory property will be used to locate the `LoggerContextFactory`. In Log4j 2 this is provided by `Log4jContextFactory`.

Applications may change the LoggerContextFactory that will be used by

1. Implementing a new `LoggerContextFactory` and creating a `log4j-provider.properties` to reference it making sure that it has the highest priority.
2. Create a new `log4j-provider.xml` and configure it with the desired `LoggerContextFactory` making sure that it has the highest priority.
3. Setting the system property log4j2.loggerContextFactory to the name of the `LoggerContextFactory` class to use.
4. 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.

15.1.2 ContextSelector

ContextSelectors are called by the Log4j LoggerContext factory. 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.

15.1.3 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", "*"};

    /**
     * Return the Configuration.
     * @param source The InputSource.
     * @return The Configuration.
     */
    public Configuration getConfiguration(InputSource source) {
        return new XMLConfiguration(source, configFile);
    }

    /**
     * Returns the file suffixes for XML files.
     * @return An array of File extensions.
     */
    public String[] getSupportedTypes() {
        return SUFFIXES;
    }
}
```

15.1.4 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
                                            @PluginAttribute(value = "level", defaultStringValue = "ERROR") L
                                            @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);
    }
}
```

15.1.5 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.

15.1.6 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

```
@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);
    }
}
```

15.1.7 Filters

As might be expected, Filters are the 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.

```
@Plugin(name = "ThresholdFilter", category = "Core", elementType = "filter", printObject = true)
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 = "NEUT
                                               @PluginAttribute(value = "onMismatch", defaultStringValue = "D
        return new ThresholdFilter(level, onMatch, onMismatch);
    }
}
```

15.1.8 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 OutputStreamAppender {

    private StubAppender(String name, Layout layout, Filter filter, StubManager manager,
                         boolean ignoreExceptions) {
    }

    @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, manager, ignoreExceptions);
    }
}
```

15.1.9 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")
        return new SampleLayout(locationInfo, properties, complete, charset);
    }
}
```

15.1.10 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);
    }
}
```

15.1.11 Custom Plugins

See the Plugins section of the manual.

16 Extending Log4j Configuration

16.1 Custom Configurations

Log4j 2 provides a few ways for applications to create their own custom configurations.

16.1.1 Add a New Configuration via ConfigurationFactory

The easiest way to create a custom Configuration 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, nul
              null,null, null);
        final Appender appender = FileAppender.createAppender("target/test.log", "false", "false", "File", "t
              "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);
    }
}
```

16.1.2 Programatically Adding to the Current Configuration

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.

```
        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 ref = AppenderRef.createAppenderRef("File", null, null);
        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 Custom Log Levels

17.1 Custom Log Levels

17.1.1 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:

```
// 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.

Standard Level	intLevel
OFF	0
FATAL	100
ERROR	200
WARN	300
INFO	400
DEBUG	500
TRACE	600
ALL	`Integer.MAX_VALUE`

Standard log levels built-in to Log4J

17.1.2 Defining Custom Log Levels in Configuration

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.

Parameter Name	Type	Description
name	String	The name of the custom level. Note that level names are case sensitive. The convention is to use all upper-case names.
intLevel	integer	Determines where the custom level exists in relation to the standard levels built-in to Log4J 2 (see the table above).

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

17.1.3 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 fourteen `debug()` methods that support the DEBUG level:

```
// 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)
```

Similar methods exist for the other built-in levels. Custom levels, in contrast, need to pass in the log level as an extra parameter.

```
// 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:

```
// nice to have: descriptive methods and no need to pass the level as a parameter
logger.verbose("a verbose message");
logger.diag("another message");
```

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.

17.1.4 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.

17.1.5 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.

17.1.6 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 someMethod() {
        // ...
        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");
        // ...
    }
    ...
}
```

17.1.7 Generating Extended Loggers

Use the following command to generate a logger wrapper that adds methods to the built-in ones:

```
java -cp log4j-core-${Log4jReleaseVersion}.jar org.apache.logging.log4j.core.tools.Generate$ExtendedLogger} \
        com.mycomp.ExtLogger DIAG=350 NOTICE=450 VERBOSE=550 > com/mycomp/ExtLogger.java
```

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.

17.1.8 Generating Custom Loggers

Use the following command to generate a logger wrapper that hides the built-in levels and has only custom levels:

```
java -cp log4j-core-${Log4jReleaseVersion}.jar org.apache.logging.log4j.core.tools.Generate$CustomLogger \
        com.mycomp.MyLogger DEFCON1=350 DEFCON2=450 DEFCON3=550 > com/mycomp/MyLogger.java
```

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.