Log4j Tutorial

LOG4J TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Log4j Tutorial

Log4j is a popular logging package written in Java.log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages.

Audience

This tutorial has been prepared for the beginners to help them understand basic functionality of log4J Logging framework. This tutorial will bring you at intermediate level of expertise from where you can take yourself at higher level of expertise.

Prerequisites

Because you are going to use log4J logging framework in various Java Based Application Development, So before proceeding with this tutorial you should have a good understanding of Java programming language.

Copyright & Disclaimer Notice

Table of Content

CHAPTER

1

Log4j Overview

Log4j is a Reliable, Fast and Flexible Logging Framework (APIs) written in Java which is distributed under the Apache Software License.

Log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages.

Log4j is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console, UNIX Syslog etc.

Log4j has three main components:

- **loggers:** Responsible for capturing logging information.
- **appenders :** Responsible for publishing logging information to various preferred destinations.
- **layouts:** Responsible to format logging information in different styles.

History of log4j

- Started in early 1996 as tracing API for the E.U. SEMPER (Secure Electronic Marketplace for Europe) project.
- 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/.

log4j Features:

- log4j is thread-safe.
- log4j is optimized for speed.

- log4j is based on a named logger hierarchy.
- log4j supports multiple output appenders per logger.
- log4j supports internationalization.
- log4j is not restricted to a predefined set of facilities.
- Logging behavior can be set at runtime using a configuration file.
- log4j is designed to handle Java Exceptions from the start.
- log4j uses multiple levels, namely ALL, TRACE, DEBUG, INFO, WARN, ERROR and FATAL.
- The format of the log output can be easily changed by extending the *Layout* class.
- The target of the log output as well as the writing strategy can be altered by implementations of the Appender interface.
- log4j is fail-stop. However, altough it certainly strives to ensure delivery, log4j does not guarantee that each log statement will be delivered to its destination.

Pros N Cons of Logging:

Logging is an important component of the software development. A well written logging code offers quick debugging, easy maintenance, and structured storage of an application's runtime information.

Logging does have its drawbacks also. 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.

CHAPTER

2

Log4j Installation

Log4j API 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 athttp://logging.apache.org/log4j/.

Once downloaded apache-log4j-x.x.x.tar.gz do the following:

Step 1:

Unzip and untar the downloaded file in /usr/local/ directory as follows:

```
$ gunzip apache-log4j-1.2.15.tar.gz
$ tar -xvf apache-log4j-1.2.15.tar
apache-log4j-1.2.15/tests/input/
apache-log4j-1.2.15/tests/input/xml/
apache-log4j-1.2.15/tests/src/
apache-log4j-1.2.15/tests/src/java/
apache-log4j-1.2.15/tests/src/java/org/
.......................................
```

While untarring, it would create a directory hierarchy with a name apache-log4j-x.x.x as follows:

```
-rw-r--r--  1 root root   3565 2007-08-25 00:09 BUILD-INFO.txt
-rw-r--r--  1 root root   2607 2007-08-25 00:09 build.properties.sample
-rw-r--r--  1 root root  32619 2007-08-25 00:09 build.xml
drwxr-xr-x 14 root root   4096 2010-02-04 14:09 contribs
drwxr-xr-x  5 root root   4096 2010-02-04 14:09 examples
-rw-r--r--  1 root root   2752 2007-08-25 00:09 INSTALL
-rw-r--r--  1 root root   4787 2007-08-25 00:09 KEYS
-rw-r--r--  1 root root  11366 2007-08-25 00:09 LICENSE
-rw-r--r--  1 root root 391834 2007-08-25 00:29 log4j-1.2.15.jar
-rw-r--r--  1 root root    160 2007-08-25 00:09 NOTICE
-rwxr-xr-x  1 root root  10240 2007-08-25 00:27 NTEventLogAppender.dll
-rw-r--r--  1 root root  17780 2007-08-25 00:09 pom.xml
drwxr-xr-x  7 root root   4096 2007-08-25 00:13 site
drwxr-xr-x  8 root root   4096 2010-02-04 14:08 src
drwxr-xr-x  6 root root   4096 2010-02-04 14:09 tests
```

Step 2:

This step is optional and depends on what features you are going to use from log4j framework. If you already have following packages installed on your machine then its fine, otherwise you would need to install them to make log4j work.

- **JavaMail API:** The e-mail.based logging feature in log4j requires the Java Mail API (mail.jar) to be installed on your machine from https://glassfish.dev.java.net/javaee5/mail/.
- **JavaBeans Activation Framework:** The Java Mail API will also require that the JavaBeans Activation Framework (activation.jar) be installed on your machine fromhttp://java.sun.com/products/javabeans/jaf/index.jsp.
- **Java Message Service:** The JMS-compatible features of log4j will require that both JMS and JNDI (Java Naming and Directory Interface) be installed on your machine fromhttp://java.sun.com/products/jms.
- **XML Parser:** You need a JAXP-compatible XML parser to use log4j. Make sure you have Xerces.jar installed on your machine from http://xerces.apache.org/xerces-j/install.html.

Step 3:

This step is very important which needs to setup CLASSPATH and PATH variables appropriately. Here I'm going to set it just for log4j.x.x.x.jar file.

```
$ pwd
/usr/local/apache-log4j-1.2.15
$ export CLASSPATH= \
      $CLASSPATH:/usr/local/apache-log4j-1.2.15/log4j-1.2.15.jar
$ export PATH=$PATH:/usr/local/apache-log4j-1.2.15/
```

CHAPTER

3

Log4j Architecture

Log4j API has been designed in layered where each layer provides different object which performs different tasks. This makes design flexible and very much extendable in future based on need.

There are two type of objects available with Log4j framework.

- **Core Objects:** These are mandatory objects of the framework and required to use the framework.
- **Support Objects:** These are optional objects of the framework and support core objects to perform addition but important tasks.

Core Objects:

LOGGER OBJECT:

The top level layer is Logger which provides Logger object. The Logger object is responsible for capturing logging information and they are stored in a namespace hierarchy.

LAYOUT OBJECT:

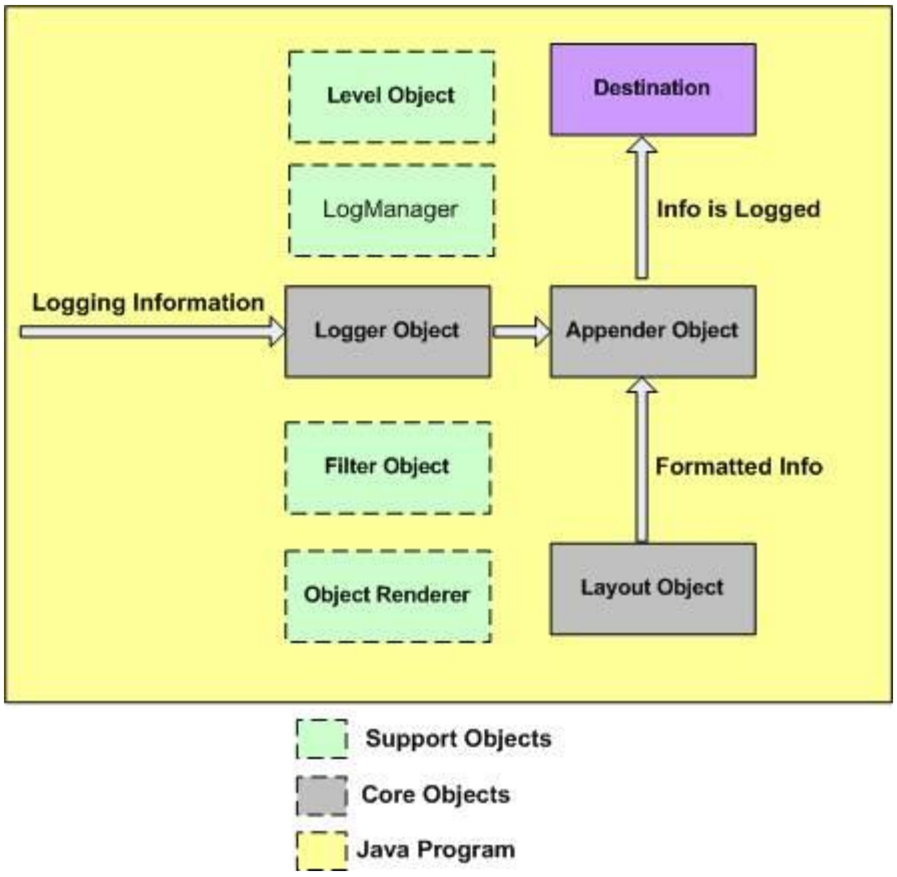
The layer provides objects which are used to format logging information in different styles. Layout layer provides support to appender objects to before publishing logging information.

Layout objects play an important role in publishing logging information in a way that is human-readable and reusable.

APPENDER OBJECT:

This is lower level layer which provides Appender object. The Appender object is responsible for publishing logging information to various preferred destinations such as a database, file, console, UNIX Syslog etc.

Following is a virtual diagram showing different components of Log4J Framework:



Support Objects:

There are other important objects in the log4j framework that play a vital role in the logging framework:

LEVEL OBJECT:

The Level object defines the granularity and priority of any logging information. There are seven levels of logging defined within the API: OFF, DEBUG, INFO, ERROR, WARN, FATAL, and ALL.

FILTER OBJECT:

The Filter object is used to analyze logging information and make further decisions on whether that information should be logged or not.

An Appender objects can have several Filter objects associated with them. If logging information is passed to a particular Appender object, all the Filter objects associated with that Appender need to approve the logging information before it can be published to the attached destination.

OBJECTRENDERER:

The ObjectRenderer object is specialized in providing a String representation of different objects passed to the logging framework. This object is used by Layout objects to prepare the final logging information.

LOGMANAGER:

The LogManager object manages the logging framework. It is responsible for reading the initial configuration parameters from a system-wide configuration file or a configuration class.

CHAPTER

4

Log4j Configuration

Previous chapter explained core components of log4j. This chapter explains how you can configure these core components using a configuration file. Configuring log4j involves assigning the Level, defining Appender, and specifying Layout objects in a configuration file.

The *log4j.properties* file is a log4j configuration file which keeps properties in key-value pairs. By default, the LogManager looks for a file named *log4j.properties* in the CLASSPATH.

- The level of the root logger is defined as DEBUG and attaches appender named X to it.
- Set the appender named X to be a valid appender.
- Set the layout for the appender X

log4j.properties Syntax:

Following is the syntax of *log4j.properties* file for an appender X:

```
# Define the root logger with appender X
log4j.rootLogger = DEBUG, X

# Set the appender named X to be a File appender
log4j.appender.X=org.apache.log4j.FileAppender

# Define the layout for X appender
log4j.appender.X.layout=org.apache.log4j.PatternLayout
log4j.appender.X.layout.conversionPattern=%m%n
```

log4j.properties Example:

Using the above syntax, we define the following in *log4j.properties* file:

- The level of the root logger is defined as DEBUG and attaches appender named FILE to it.
- The appender FILE is defined as org.apache.log4j.FileAppender and writes to a file named "log.out" located in the **log** directory.
- The layout pattern defined is %m%n, which means the printed logging message will be followed by a newline character.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

It is important to note that log4j supports UNIX-style variable substitution such as ${variableName}.

Debug Level:

We have used DEBUG with both the appenders. All the possible options are:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- ALL

These levels would be explained in Log4j Logging Levels.

Appenders:

Apache log4j provides Appender objects which are primarily responsible for printing logging messages to different destinations such as consoles, files, sockets, NT event logs, etc.

Each Appender object has different properties associated with it, and these properties indicate the behavior of that object.

Property	**Description**
layout	Appender uses the Layout objects and the conversion pattern associated with them to format the logging information.
target	The target may be a console, a file, or another item depending on the appender.
level	The level is required to control the filteration of the log messages.
threshold	Appender can have a threshold level associated with it independent of the logger level. The Appender ignores any logging messages that have a level lower than the threshold level.
filter	The Filter objects can analyze logging information beyond level matching and decide whether logging requests should be handled by a particular Appender or ignored.

We can add an Appender object to a Logger by including the following setting in the configuration file with the following method:

```
log4j.logger.[logger-name]=level, appender1,appender..n
```

You can write same configuration in XML format as follows:

```
<logger name="com.apress.logging.log4j" additivity="false">
   <appender-ref ref="appender1"/>
   <appender-ref ref="appender2"/>
</logger>
```

If you are willing to add Appender object inside your program then you can use following method:

```
public void addAppender(Appender appender);
```

The addAppender() method adds an Appender to the Logger object. As the example configuration demonstrates, it is possible to add many Appender objects to a logger in a comma-separated list, each printing logging information to separate destinations.

We have used only one appender *FileAppender* in our example above. All the possible appender options are:

- AppenderSkeleton
- AsyncAppender
- ConsoleAppender
- DailyRollingFileAppender
- ExternallyRolledFileAppender
- FileAppender
- JDBCAppender
- JMSAppender
- LF5Appender
- NTEventLogAppender
- NullAppender
- RollingFileAppender
- SMTPAppender
- SocketAppender
- SocketHubAppender
- SyslogAppender
- TelnetAppender

- WriterAppender

We would cover FileAppender in Logging in Files and JDBCAppender would be convered in Logging in Database.

Layout:

We have used PatternLayout with our appender. All the possible options are:

- DateLayout
- HTMLLayout
- PatternLayout
- SimpleLayout
- XMLLayout

Using HTMLLayout and XMLLayout you can generate log in HTML and in XML format as well.

Layout Formatting:

You would learn how to format a log message in chapter:Log Formatting.

CHAPTER

5

Log4j Sample Program

We have seen how to create a configuration file. This tutorial would teach you how to generate debug messages and log them in a simple text file.

Following is a simple configuration file created for our example. Let me re-iterate it once again:

- The level of the root logger is defined as DEBUG and attaches appender named FILE to it.
- The appender FILE is defined as org.apache.log4j.FileAppender and writes to a file named "log.out" located in the **log** directory.
- The layout pattern defined is %m%n, which means the printed logging message will be followed by a newline character.

So the content of *log4j.properties* file would be as follows:

```
# Define the root logger with appender file
log = /usr/home/log4j
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Using log4j in Java Program:

The following Java class is a very simple example that initializes, and then uses, the Log4J logging library for Java applications.

```
import org.apache.log4j.Logger;

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class log4jExample{
  /* Get actual class name to be printed on */
  static Logger log = Logger.getLogger(
                      log4jExample.class.getName());

  public static void main(String[] args)
                throws IOException,SQLException{

     log.debug("Hello this is an debug message");
     log.info("Hello this is an info message");
  }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned program. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

All the libraries should be available in CLASSPATH and your *log4j.properties* file should be available in PATH. So do the following:

- Create log4j.properties as shown above.
- Create log4jExample.java as shown above and compile it.
- Execute log4jExample binary to run the program.

You would get following result, inside /usr/home/log4j/log.out file:

```
Hello this is an debug message
Hello this is an info message
```

CHAPTER

6

Log4j Logging Methods

Logger class provides a variety of methods to handle logging activities. The Logger class does not allow us to instantiate a new Logger instance but it provides two static methods for obtaining a Logger object:

- **public static Logger getRootLogger();**
- **public static Logger getLogger(String name);**

Here the first of the two methods returns the application instance's root logger and does not have a name. Any other named Logger object instance is obtained through the second method by passing the name of the logger. The name of the logger can be any string you can pass, usually class or package name as we have used in last chapter and it is mentioned blow:

```
static Logger log = Logger.getLogger(log4jExample.class.getName());
```

Logging Methods:

Once we obtain an instance of a named logger, we can use several methods of the logger to log messages. The Logger class has the following methods for printing the logging information.

SN	Methods with Description
1	**public void debug(Object message)** This method prints messages with the level Level.DEBUG.
2	**public void error(Object message)** This method prints messages with the level Level.ERROR.
3	**public void fatal(Object message);** This method prints messages with the level Level.FATAL.
4	**public void info(Object message);** This method prints messages with the level Level.INFO.
5	**public void warn(Object message);** This method prints messages with the level Level.WARN.
6	**public void trace(Object message);** This method prints messages with the level Level.TRACE.

All the levels are defined in the org.apache.log4j.Level class and any of the above mentioned method can be called as follows:

```
import org.apache.log4j.Logger;

public class LogClass {
   private static org.apache.log4j.Logger log = Logger
                                    .getLogger(LogClass.class);
   public static void main(String[] args) {
      log.trace("Trace Message!");
      log.debug("Debug Message!");
      log.info("Info Message!");
      log.warn("Warn Message!");
      log.error("Error Message!");
      log.fatal("Fatal Message!");
   }
}
```

When you compile and run LogClass program it would generate following result:

```
Debug Message!
Info Message!
Warn Message!
Error Message!
Fatal Message!
```

All the debug messages make more sense when they are used in combination of levels. I would cover level in next chapter and then you would have good understanding on how to use these methods in combination of different levels of debugging.

CHAPTER

7

Log4j Logging Levels

The org.apache.log4j.Level class provides following levels but you can also define your custom levels by sub-classing the Level class.

Level	Description
ALL	All levels including custom levels.
DEBUG	Designates fine-grained informational events that are most useful to debug an application.
ERROR	Designates error events that might still allow the application to continue running.
FATAL	Designates very severe error events that will presumably lead the application to abort.
INFO	Designates informational messages that highlight the progress of the application at coarse-grained level.
OFF	The highest possible rank and is intended to turn off logging.
TRACE	Designates finer-grained informational events than the DEBUG.
WARN	Designates potentially harmful situations.

How Level Works?

A log request of level p in a logger with level q, is enabled if p >= q. This rule is at the heart of log4j. It assumes that levels are ordered. For the standard levels, we have ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.

Following example makes it clear that how we can filter our all the DEBUG and INFO messages. This program makes use of logger method setLevel(Level.X) to set a desired logging level:

This example would print all the messages except Debug and Info:

```
import org.apache.log4j.*;


public class LogClass {

   private static org.apache.log4j.Logger log = Logger
```

```
                                  .getLogger(LogClass.class);
   public static void main(String[] args) {
      log.setLevel(Level.WARN);

      log.trace("Trace Message!");
      log.debug("Debug Message!");
      log.info("Info Message!");
      log.warn("Warn Message!");
      log.error("Error Message!");
      log.fatal("Fatal Message!");
   }
}
```

When you compile and run LogClass program it would generate following result:

```
Warn Message!
Error Message!
Fatal Message!
```

Setting Levels using Configuration File:

Log4j provides you configuration file based level setting which puts you free from changing source code when you want to change debugging level.

Following is an example configuration file which would do the same as we did using*log.setLevel(Level.WARN)* method in the above example.

```
# Define the root logger with appender file
log = /usr/home/log4j
log4j.rootLogger = WARN, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Now let us use our following program:

```
import org.apache.log4j.*;


public class LogClass {
   private static org.apache.log4j.Logger log = Logger
                                    .getLogger(LogClass.class);
   public static void main(String[] args) {
      log.trace("Trace Message!");
      log.debug("Debug Message!");
      log.info("Info Message!");
      log.warn("Warn Message!");
      log.error("Error Message!");
      log.fatal("Fatal Message!");
   }
}
```

Now compile and run above program and you would get following result in /usr/home/log4j/log.out file:

```
Warn Message!
Error Message!
Fatal Message!
```

CHAPTER

8

Log4j Log Formatting

Apache log4j provides various Layout objects, each of which can format logging data according to various layouts. It is also possible to create a Layout object that formats logging data in an application-specific way.

All Layout objects receive a LoggingEvent object from the Appender objects. The Layout objects then retrieve the message argument from the LoggingEvent and apply the appropriate ObjectRenderer to obtain the String representation of the message.

The Layout Types:

The top-level class in the hierarchy is the abstract class org.apache.log4j.Layout. This is the base class for all other Layout classes in the log4j API.

The Layout class is defined as abstract within an application, we never use this class directly; instead, we work with its subclasses which are as follows:

- DateLayout
- HTMLLayout (Explained in this tutorial)
- PatternLayout. (Explained in this tutorial)
- SimpleLayout
- XMLLayout

HTML Layout

If you want to generate your logging information in a HTML-formatted file then you can use org.apache.log4j.HTMLLayout to format your logging information.

The HTMLLayout class extends the abstract org.apache.log4j.Layout class and overrides the format() method from its base class to provide HTML-style formatting.

This provided the following information to be displayed:

- The time elapsed from the start of the application before a particular logging event was generated.

- The name of the thread that invoked the logging request.
- The level associated with this logging request.
- The name of the logger and logging message.
- The optional location information for the program file and the line number from which this logging was invoked.

HTMLLayout is a very simple Layout object that provides following methods:

S.N.	Method & Description
1	**setContentType(String)** Sets the content type of the text/html HTML content. Default is text/html.
2	**setLocationInfo(String)** Sets the location information for the logging event. Default is false.
3	**setTitle(String)** Sets the title for the HTML file. Default is Log4j Log Messages.

HTMLLayout Example:

Following is a simple configuration file for HTMLLayout:

```
# Define the root logger with appender file
log = /usr/home/log4j
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/htmlLayout.html

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.HTMLLayout
log4j.appender.FILE.layout.Title=HTML Layout Example
log4j.appender.FILE.layout.LocationInfo=true
```

Now consider the following Java Example which would generate logging information:

```
import org.apache.log4j.Logger;

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class log4jExample{
  /* Get actual class name to be printed on */
  static Logger log = Logger.getLogger(
                      log4jExample.class.getName());

  public static void main(String[] args)
                throws IOException,SQLException{

      log.debug("Hello this is an debug message");
     log.info("Hello this is an info message");
  }
}
```

Compile and run the above program, it would create a htmlLayout.html file in /usr/home/log4j directory which would have following log information:

Log session start time Mon Mar 22 13:30:24 AST 2010

Time	Thread	Level	Category	File:Line	Message
0	main	DEBUG	log4jExample	log4jExample.java:15	Hello this is an debug message
6	main	INFO	log4jExample	log4jExample.java:16	Hello this is an info message

You would use a web browser to open htmlLayout.html file. It is also important to note that the footer for the </html> and </body> tags is completely missing.

One of the big advantages of having the log file in HTML format is that it can be published as a web page for remote viewing.

Pattern Layout

If you want to generate your logging information in a particular format based on a patten then you can use org.apache.log4j.PatternLayout to format your logging information.

The PatternLayout class extends the abstract org.apache.log4j.Layout class and overrides the format() method to structure logging information according to a supplied pattern.

PatternLayout is also a simple Layout object that provides following *Bean Property* which can be set using configuration file:

S.N.	Property & Description
1	**conversionPattern** Sets the conversion pattern. Default is %r [%t] %p %c %x - %m%n

Pattern Conversion Characters:

Below table explains characters used in the above pattern and all other characters which you can use in your custom pattern:

Conversion Character	Meaning
c	Used to output the category of the logging event. For example, for the category name "a.b.c" the pattern %c{2} will output "b.c".
C	Used to output the fully qualified class name of the caller issuing the logging request. For example, for the class name "org.apache.xyz.SomeClass", the pattern %C{1} will output "SomeClass".
d	Used to output the date of the logging event. For example, %d{HH:mm:ss,SSS} or %d{dd MMM yyyy HH:mm:ss,SSS}.
F	Used to output the file name where the logging request was issued.
l	Used to output location information of the caller which generated the logging event.
L	Used to output the line number from where the logging request was issued.
m	Used to output the application supplied message associated with the logging event.
M	Used to output the method name where the logging request was issued.
n	Outputs the platform dependent line separator character or characters.

p	Used to output the priority of the logging event.
r	Used to output the number of milliseconds elapsed from the construction of the layout until the creation of the logging event.
t	Used to output the name of the thread that generated the logging event.
x	Used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event.
X	The X conversion character is followed by the key for the MDC. For example, X{clientIP} will print the information stored in the MDC against the key clientIP.
%	The literal percent sign. %% will print a % sign.

Format Modifiers:

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.

Following table covers various modifiers scenarios:

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, hen 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.

PatternLayout Example:

Following is a simple configuration file for PatternLayout:

```
# Define the root logger with appender file
log = /usr/home/log4j
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=
      %d{yyyy-MM-dd}-%t-%x-%-5p-%-10c:%m%n
```

Now consider the following Java Example which would generate logging information:

```
import org.apache.log4j.Logger;

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class log4jExample{
  /* Get actual class name to be printed on */
  static Logger log = Logger.getLogger(
                      log4jExample.class.getName());

  public static void main(String[] args)
                throws IOException,SQLException{

     log.debug("Hello this is an debug message");
     log.info("Hello this is an info message");
  }
}
```

Compile and run the above program, it would create a log.out file in /usr/home/log4j directory which would have following log information:

```
2010-03-23-main--DEBUG-log4jExample:Hello this is an debug message
2010-03-23-main--INFO -log4jExample:Hello this is an info message
```

The Layout Methods:

This class provides a skeleton implementation of all the common operations across all other Layout objects and declares two abstract methods.

S.N.	Method & Description
1	**public abstract boolean ignoresThrowable()** This method indicates whether the logging information handles any java.lang.Throwable object passed to it as a part of the logging event. If the Layout object handles the Throwable object, then the Layout object does not ignore it, and returns false.
2	**public abstract String format(LoggingEvent event)** Individual layout subclasses will implement this method for layout specific formatting.

Apart from these abstract methods, the Layout class provides concrete implementation for the methods listed below:

S.N.	Method & Description
1	**public String getContentType()** Returns the content type used by the Layout objects. The base class returns text/plain as the default content type.
2	**public String getFooter()** Specifies the footer information of the logging message.
3	**public String getHeader()** Specifies the header information of the logging message.

Each subclass can return class-specific information by overriding the concrete implementation of these methods.

CHAPTER

9

Log4j Logging in Files

To write your logging information into a file you would have to use *org.apache.log4j.FileAppender*. There are following configurable parameters of FileAppender:

FileAppender Configuration:

Property	**Description**
immediateFlush	This flag is by default set to true, which means the output stream to the file being flushed with each append operation.
encoding	It is possible to use any character-encoding. By default is the platform-specific encoding scheme.
threshold	The threshold level for this appender.
Filename	The name of the log file.
fileAppend	This is by default set to true, which mean the logging information being appended to the end of the same file.
bufferedIO	This flag indicates whether we need buffered writing enabled. By default is set to false.
bufferSize	If bufferedI/O is enabled, this indicates the buffer size. By default is set to 8kb.

Following is a sample configuration file *log4j.properties* for FileAppender.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
# Set the name of the file
log4j.appender.FILE.File=${log}/log.out

# Set the immediate flush to true (default)
log4j.appender.FILE.ImmediateFlush=true

# Set the threshold to debug mode
log4j.appender.FILE.Threshold=debug

# Set the append to false, overwrite
```

```
log4j.appender.FILE.Append=false

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

If you love to have an XML configuration file equivalent to above *log4j.properties* file then here is the content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>

<appender name="FILE" class="org.apache.log4j.FileAppender">
   <param name="file" value="${log}/log.out"/>
   <param name="immediateFlush" value="true"/>
   <param name="threshold" value="debug"/>
   <param name="append" value="false"/>
   <layout class="org.apache.log4j.PatternLayout">
      <param name="conversionPattern" value="%m%n"/>
   </layout>
</appender>

<logger name="log4j.rootLogger" additivity="false">
   <level value="DEBUG"/>
   <appender-ref ref="FILE"/>
</logger>

</log4j:configuration>
```

You can try log4j - Sample Program with the above configuration .

Logging in Multiple Files:

There may be a requirement when you want to write your log message into multiple files for certain reasons like for example if file size reaches to a certain threshold etc.

To write your logging information into multiple files you would have to useorg.apache.log4j.RollingFileAppender class which extends the *FileAppender* class and inherits all its properties.

There are following configurable parameters in addition to what have been mentioned above for FileAppender:

Property	**Description**
maxFileSize	This is the critical size of the file above which the file will be rolled. Default value is 10MB
maxBackupIndex	This property denotes the number of backup files to be created. Default value is 1.

Following is a sample configuration file *log4j.properties* for RollingFileAppender.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.RollingFileAppender
# Set the name of the file
log4j.appender.FILE.File=${log}/log.out

# Set the immediate flush to true (default)
log4j.appender.FILE.ImmediateFlush=true
```

```
# Set the threshold to debug mode
log4j.appender.FILE.Threshold=debug

# Set the append to false, should not overwrite
log4j.appender.FILE.Append=true

# Set the maximum file size before rollover
log4j.appender.FILE.MaxFileSize=5KB

# Set the the backup index
log4j.appender.FILE..MaxBackupIndex=2

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

If you like to have an XML configuration file, you can generate as mentioned in initial section and add only additional paramters related to *RollingFileAppender*.

This example configuration demonstrates that maximum permissible size of each log file is 5MB. Upon exceeding the maximum size, a new log file will be created and because *maxBackupIndex* is defined as 2, once the second log file reaches the maximum size, the first log file will be erased and thereafter all the logging information will be rolled back to the first log file.
You can try log4j - Sample Program with the above configuration .

Daily Log File Generation:

There may be a requirement when you want to generate your log files on per day basis to keep a clean record of your logging information.

To write your logging information into files on daily basis you would have to use*org.apache.log4j.DailyRollingFileAppender* class which extends the *FileAppender* class and inherits all its properties.

There is only one important following configurable parameter in addition to what have been mentioned above for FileAppender:

Property	Description
DatePattern	This indicates when to roll over the file, and the naming convention to be followed. By default roll over at midnight each day.

DatePattern controls the rollover schedule using one of the following patterns:

DatePattern	Description
'.' yyyy-MM	Roll over at the end of each month and the beginning of the next month.
'.' yyyy-MM-dd	This is the default value and roll over at midnight each day.
'.' yyyy-MM-dd-a	Roll over at midday and midnight of each day.
'.' yyyy-MM-dd-HH	Roll over at the top of every hour.
'.' yyyy-MM-dd-HH-mm	Roll over every minute.
'.' yyyy-ww	Roll over on the first day of each week depending upon the locale.

Following is a sample configuration file *log4j.properties* to generate log files rolling over at midday and midnight of each day.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.DailyRollingFileAppender
# Set the name of the file
log4j.appender.FILE.File=${log}/log.out

# Set the immediate flush to true (default)
log4j.appender.FILE.ImmediateFlush=true

# Set the threshold to debug mode
log4j.appender.FILE.Threshold=debug

# Set the append to false, should not overwrite
log4j.appender.FILE.Append=true

# Set the DatePattern
log4j.appender.FILE.DatePattern='.' yyyy-MM-dd-a

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

If you like to have an XML configuration file, you can generate as mentioned in initial section and add only additional paramters related to *DailyRollingFileAppender*.

You can try log4j - Sample Program with the above configuration .

CHAPTER

10

Log4j Logging in Database

The log4j API provides the *org.apache.log4j.jdbc.JDBCAppender* object, which is capable of putting logging information in a specified database.

JDBCAppender Configuration:

Property	Description
bufferSize	Sets the buffer size. Default size is 1.
Driver	Sets the driver class to the specified string. If no driver class is specified, defaults to sun.jdbc.odbc.JdbcOdbcDriver.
Layout	Sets the layout to be used. Default layout is org.apache.log4j.PatternLayout.
Password	Sets the database password.
Sql	Specifies SQL statement to be executed every time a logging event occurs. This could be INSERT, UPDATE, or DELETE.
URL	Sets the JDBC URL
User	Sets the database user name

Log Table Configuration:

Before you start using JDBC based logging, you shold create a table where all the log information would be maintained. Following is the SQL Statement for Creating the LOGS Table:

```
CREATE TABLE LOGS
   (USER_ID VARCHAR(20) NOT NULL,
    DATED   DATE NOT NULL,
    LOGGER  VARCHAR(50) NOT NULL,
    LEVEL   VARCHAR(10) NOT NULL,
    MESSAGE VARCHAR(1000) NOT NULL
   );
```

Sample Configuration File:

Following is a sample configuration file *log4j.properties* for JDBCAppender which will be used to log messages to a LOGS table.

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, DB

# Define the DB appender
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender

# Set JDBC URL
log4j.appender.DB.URL=jdbc:mysql://localhost/DBNAME

# Set Database Driver
log4j.appender.DB.driver=com.mysql.jdbc.Driver

# Set database user name and password
log4j.appender.DB.user=user_name
log4j.appender.DB.password=password

# Set the SQL statement to be executed.
log4j.appender.DB.sql=INSERT INTO LOGS
                      VALUES('%x','%d','%C','%p','%m')

# Define the layout for file appender
log4j.appender.DB.layout=org.apache.log4j.PatternLayout
```

Here for a MySQL database, you would have to use actual DBNAME, user id and password where you have created LOGS table. The SQL statement is to execute an INSERT statement with the table name LOGS and values to be entered into the table.

The JDBCAppender does not need a layout to be defined explicitly. Instead, the SQL statement passed to it uses a PatternLayout

If you like to have an XML configuration file equivalent to above *log4j.properties* file, then here is the content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>

<appender name="DB" class="org.apache.log4j.jdbc.JDBCAppender">
   <param name="url" value="jdbc:mysql://localhost/DBNAME"/>
   <param name="driver" value="com.mysql.jdbc.Driver"/>
   <param name="user" value="user_id"/>
   <param name="password" value="password"/>
   <param name="sql" value="INSERT INTO LOGS VALUES('%x',
                             '%d','%C','%p','%m')"/>
   <layout class="org.apache.log4j.PatternLayout">
   </layout>

</appender>

<logger name="log4j.rootLogger" additivity="false">
   <level value="DEBUG"/>
   <appender-ref ref="DB"/>
</logger>

</log4j:configuration>
```

Sample Program:

The following Java class is a very simple example that initializes, and then uses, the Log4J logging library for Java applications.

```
import org.apache.log4j.Logger;
import java.sql.*;
import java.io.*;
import java.util.*;

public class log4jExample{
  /* Get actual class name to be printed on */
  static Logger log = Logger.getLogger(
                      log4jExample.class.getName());

  public static void main(String[] args)
                throws IOException,SQLException{

     log.debug("Debug");
     log.info("Info");
  }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned program. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

All the libraries should be available in CLASSPATH and your *log4j.properties* file should be available in PATH. So do the following:

- Create log4j.properties as shown above.
- Create log4jExample.java as shown above and compile it.
- Execute log4jExample binary to run the program.

Now check your LOGS table inside DBNAME database and you would find following entries:

```
mysql >  select * from LOGS;
+---------+------------+--------------+-------+---------+
| USER_ID | DATED      | LOGGER       | LEVEL | MESSAGE |
+---------+------------+--------------+-------+---------+
|         | 2010-05-13 | log4jExample | DEBUG | Debug   |
|         | 2010-05-13 | log4jExample | INFO  | Info    |
+---------+------------+--------------+-------+---------+
2 rows in set (0.00 sec)
```

NOTE: Here x is used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event. We use NDC to distinguish clients in server-side components handling multiple clients. Check Log4J Manual for more information on this.