log4j

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 for formatting 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 and several incarnations, the initial API has evolved to become log4j, a popular logging package for Java.
* The package is distributed under the Apache Software License, a full-fledged open source license certified by the open source initiative.
* The latest log4j version, including its full-source code, class files, and documentation can be found at <http://logging.apache.org/log4j/>.

log4j Features

* It is thread-safe.
* It is optimized for speed.
* It is based on a named logger hierarchy.
* It supports multiple output appenders per logger.
* It supports internationalization.
* It is not restricted to a predefined set of facilities.
* Logging behavior can be set at runtime using a configuration file.
* It is designed to handle Java Exceptions from the start.
* It 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.
* It is fail-stop. However, although it certainly strives to ensure delivery, log4j does not guarantee that each log statement will be delivered to its destination.

Pros and 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.

Configuration of Log4j

* 1. Create maven project
  2. Go to pom.xml file and add dependency, version suited as per your system

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

* 1. Add “log4j.properties” file in src/main/resources under some folder. In this project it is kept as src/main/resources/ConfigFiles
  2. Open “log4j.properties” file and set paths as per your convenience
  3. Now, Add Logs.java file in which functions is predefined. You can also customize functions and add your methods too. In this project Logs file is kept under - TestNG0015Log4j -> Logs

Implementation of Log4j

* 1. Create any type of listener as per your requirements. I took ITestListener.
  2. Integrate property file in that listener class and do few changes. It is given in TestNG0015Log4j -> ListenerITestListenerWithLog4j.java
  3. We can also integrate this on test level, suite level or any level we want.
  4. Call listener class, in your class where your actual test case written
  5. Finally run the test and refresh the project and see the result.

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);**

The first of the two methods returns the application instance's root logger and it 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 a class or a package name as we have used in the last chapter and it is mentioned below −

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.

|  |  |
| --- | --- |
| **#** | **Methods and Description** |
| 1 | **public void debug(Object message)**  It prints messages with the level Level.DEBUG. |
| 2 | **public void error(Object message)**  It prints messages with the level Level.ERROR. |
| 3 | **public void fatal(Object message)**  It prints messages with the level Level.FATAL. |
| 4 | **public void info(Object message)**  It prints messages with the level Level.INFO. |
| 5 | **public void warn(Object message)**  It prints messages with the level Level.WARN. |
| 6 | **public void trace(Object message)**  It 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 methods 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 the 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 with levels. We will cover levels in the next chapter and then, you would have a good understanding of how to use these methods in combination with different levels of debugging.

The **org.apache.log4j.Level** levels. 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. |
| INFO | Designates informational messages that highlight the progress of the application at coarse-grained level. |
| WARN | Designates potentially harmful situations. |
| 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. |
| OFF | The highest possible rank and is intended to turn off logging. |
| TRACE | Designates finer-grained informational events than the DEBUG. |

## How do Levels 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.

The Following example shows how we can filter all our DEBUG and INFO messages. This program uses 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 the **LogClass** program, it would generate the following result −

Warn Message!

Error Message!

Fatal Message!

## Setting Levels using Configuration File

log4j provides you configuration file based level setting which sets you free from changing the source code when you want to change the debugging level.

Following is an example configuration file which would perform the same task as we did using the **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

Let us now 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 the above program and you would get following result in **/usr/home/log4j/log.out** file −

Warn Message!

Error Message!

Fatal Message!

# **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](https://www.tutorialspoint.com/log4j/log4j_htmllayout.htm)
* [PatternLayout.](https://www.tutorialspoint.com/log4j/log4j_patternlayout.htm)
* SimpleLayout
* XMLLayout

## The Layout Methods

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

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **public abstract boolean ignoresThrowable()**  It 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 implement this method for layout specific formatting. |

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

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **public String getContentType()**  It returns the content type used by the Layout objects. The base class returns text/plain as the default content type. |
| 2 | **public String getFooter()**  It specifies the footer information of the logging message. |
| 3 | **public String getHeader()**  It specifies the header information of the logging message. |

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

# **log4j - Logging in Files**

To write your logging information into a file, you would have to use **org.apache.log4j.FileAppender**.

## FileAppender Configuration

FileAppender has the following configurable parameters:

|  |  |
| --- | --- |
| **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, it 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 means the logging information being appended to the end of the same file. |
| bufferedIO | This flag indicates whether we need buffered writing enabled. By default, it is set to false. |
| bufferSize | If buffered I/O is enabled, it indicates the buffer size. By default, it 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 wish to have an XML configuration file equivalent to the 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](http://www.tutorialspoint.com/log4j/log4j_sample_program.htm) with the above configuration.

## Logging in Multiple Files

You may want to write your log messages into multiple files for certain reasons, for example, if the file size reached to a certain threshold.

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

We have the following configurable parameters in addition to the ones 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 10 MB. |
| 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=5MB

# 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 wish to have an XML configuration file, you can generate the same as mentioned in the initial section and add only additional parameters related to **RollingFileAppender**.

This example configuration demonstrates that the maximum permissible size of each log file is 5 MB. Upon exceeding the maximum size, a new log file will be created. Since **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](http://www.tutorialspoint.com/log4j/log4j_sample_program.htm) with the above configuration.

## Daily Log File Generation

There may be a requirement to generate your log files on a daily basis to keep a clean record of your logging information.

To write your logging information into files on a 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 configurable parameter in addition to the ones 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 is performed 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 at the beginning of the next month. |
| '.' yyyy-MM-dd | Roll over at midnight each day. This is the default value. |
| '.' 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 wish to have an XML configuration file, you can generate the same as mentioned in the initial section and add only additional parameters related to **DailyRollingFileAppender**.

# **log4j - Logging in Database**

The log4j API provides the **org.apache.log4j.jdbc.JDBCAppender** object, which can put 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, it 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 the 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 should create a table to maintain all the log information. 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 is 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

For MySQL database, you would have to use the 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 the values to be entered into the table.

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

If you wish to have an XML configuration file equivalent to the 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");

}

}

## Compile and Execute

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

All the libraries should be available in **CLASSPATH** and your *log4j.properties* file should be available in PATH. Follow the given steps −

* 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 the 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 Nested diagnostic Context (NDC) 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.