UsefulCode

Please use this page to list classes or code bits related to log4j that you have found to be useful. If you have found them to be useful, chances are so will someone else.

IRC Appender

IRCAppender connects to an IRC server and joins a channel. Appended events are formatted using a layout and sent to the channel.

Requires IRClib (LGPL) available from here: http://moepii.sourceforge.net/

Appender source, class file and sample configuration are available on this page from the 'attachments' entry in the 'More actions' listbox.

Use this appender carefully (configure the appender to only send events on ERROR or FATAL, for example). The server will disconnect you if you send too many messages too quickly.

This appender won't register your nickname - connect using an IRC client and register the nick you will be using if this is necessary.

If the channel you are trying to send messages to requires you to identify yourself to nickserv, specify the 'password' param.

Because a significant amount of time may elapse between appender initialization and successfully joining a channel, events appended prior to a successful channel join are collected and sent to the channel once the join is successful. This process uses a 1 sec wait between event sends in order to prevent a flood disconnect.

If you'd like to use ChainsawV2 as an IRC client, see the ChainsawHelp wiki page.

Determining the running version of log4j:

```
import  org.apache.log4j.Layout;
public class X {
   public static void main(String[] a) {
     Package p = Layout.class.getPackage();
     System.out.println(p);
     System.out.println("Implementation title:   "+p.getImplementationTitle());
     System.out.println("Implementation vendor:  "+p.getImplementationVendor());
     System.out.println("Implementation version: "+p.getImplementationVersion());
   }
}

yields:

package org.apache.log4j
Implementation title:   log4j
Implementation vendor:  "Apache Software Foundation"
Implementation version: 1.2.8   <-- that's what you are looking for
```

It will yield the expected results with log4j 1.1.x as well as 1.2.x. Thanks to Daniel Serodio for the suggestion and Ceki for the code.

Various code bits that extend or user log4j classes and features have been submitted by many contributors. They can be found as part of the log4j release package or here.

Some (hopefully) useful filters can be found in the current log4j cvs at http://cvs.apache.org/viewcvs.cgi/logging/log4j/trunk/src/java/org/apache/log4j/filter/. They will be released, in some packaged form, with v1.3, but they are compatible with current v1.2.X.

Some custom repository selectors can be found in the current log4j cvs at http://cvs.apache.org/viewcvs.cgi/logging/log4j/trunk/src/java/org/apache/log4j/selector/. They will be released, in some packaged form, with v1.3, but they are compatible with current v1.2.X.

Web Application - Servlet related log4j classes can be found in the current log4j cvs at http://cvs.apache.org/viewcvs/jakarta-log4j-sandbox/src/java/org/apache/log4j/servlet/. They will be released, in some packaged form, with v1.3, but they are compatible with the current v1.2.X.

An **enhanced version of the configuration Servlet** actually available in cvs repository, which adds:

- Ability to sort loggers by name or level
- Easy level modification (with a select near each logger)
- Clean xhtml strict output (styled using css)
- configurable partial output (via servlet init parameter) which lets you use this servlet i portal environments or tiles

ConfigurationServlet.java

– Fabrizio Giustina

This bit of code can be used to "guess" if log4j has been previously configured. It does not determine if the configuration is valid and working as expected.

```
    /**
        Returns true if it appears that log4j have been previously configured. This code
        checks to see if there are any appenders defined for log4j which is the
        definitive way to tell if log4j is already initialized */
    private static boolean isConfigured() {
        Enumeration appenders = Logger.getRoot().getAllAppenders();
        if (appenders.hasMoreElements()) {
            return true;
        }
        else {
            Enumeration loggers = LogManager.getCurrentLoggers() ;
            while (loggers.hasMoreElements()) {
                Logger c = (Logger) loggers.nextElement();
                if (c.getAllAppenders().hasMoreElements())
                    return true;
            }
        }
        return false;
    }
```

hint: since in log4j version 1.3 Logger.getRoot() is deprecated, one might use LogManager.getRootLogger().getAllAppenders()

Or instead of guessing if log4j has been previously configured, you might want to ensure that it is configured (or shut down for that matter) only once. Here is a class that does just that. This can be particularly useful for JUnit test cases as methods configure and shutdown can be called from methods setUp() and tearDown(), respectively, of JUnit test cases.

```
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.LogManager;

public class Log4jConfigurator {

    private static boolean configured = false;

    public synchronized static void configure() {
        if (!configured) {
            BasicConfigurator.configure();
            configured = true;
        }
    }

    public synchronized static void shutdown() {
        if (configured) {
            LogManager.shutdown();
            configured = false;
        }
    }

}
```

.

Using Log4j with libs that log to a java.io.Writer, E.g. `javax.sql.DataSource`:

Thanks to Gino Marckx for suggesting a FilterWriter extension in his sample.

```
import java.io.*;
import org.apache.log4j.Logger;
import org.apache.log4j.Level;

/** A java.io.Writer implementation that forwards the stream to
 * a Log4j logger. This allows the use of Log4j with third-party
 * APIs that can only write log messages to a stream. E.g. JDBC's
 * javax.sql.DataSource implementations accept a java.io.PrintWriter
```

```
 * instance to which all log messages are written. We can forward these
 * jdbc log messages to log4j using
 * <code>
 *   dataSource.setLogWriter
 *     (new PrinterWriter
 *      (new Writer2Log4j("logger.category","debug",true)));
 * </code>.
 * How do we decide on the boundaries of messages in the stream of
 * characters? We rely on the <code>flush()</code> method.
 *
 * @revision $Revision$ $Date$
 * @author Prasad Chodavarapu
 */
public class Writer2Log4j extends FilterWriter {

    protected final Logger logger;
    protected final Level  priority;
    protected final boolean chopTerminatingNewLine;
    protected StringBuffer buffer = new StringBuffer();

    /** Constructor.
     * @arg category log4j category under which you wish messages to be logged
     * @arg priority under which messages should be logged
     * @arg chopTerminatingNewLine if set to true, terminating new
     * line chars of a log message are chopped (or chomp()ed in perl speak).
     * Most log4j conversion patterns add their own new lines (%n) as
     * part of the layout. As layouts are set for appenders and not
     * for categories, it's convenient to have this ability.
     */
    public Writer2Log4j
        (String category, String priority,
         boolean  chopTerminatingNewLine) {
        //FilterWriter, by default, writes to the stream passed
        //to its constructor. As we'll be overriding all of its methods,
        //we'll simply pass System.out but won't use it.
        super(new PrintWriter(System.out));

        logger = Logger.getLogger(category);
        this.priority = Level.toLevel(priority);
        this.chopTerminatingNewLine = chopTerminatingNewLine;
    }

    /** Appends the given character to the next log message.
     */
    public synchronized void write(int c) throws IOException {
        buffer.append(c);
    }

    /** Appends the given range of characters to the next log message.
     */
    public synchronized void
        write(char[] cbuf, int off, int len) throws IOException {
        buffer.append(cbuf, off, len);
    }


    /** Appends the given range of characters to the next log message.
     */
    public synchronized void
        write(String str, int off, int len) throws IOException {
        buffer.append(str.substring(off, off+len));
    }

    /** Flushes the buffer to log4j. */
    public synchronized void flush() throws IOException {
        log();
    }


    /** Flushes buffer to log4j if it is non-empty. */
    public synchronized void close() throws IOException {
```

```
        if (buffer.length() != 0) { log(); }
    }

    /** Flushes buffer to log4j if it is non-empty. */
    protected void finalize() throws Throwable {
        if (buffer.length() != 0) { log(); }
        super.finalize();
    }

    /** Forwards the contents of the buffer to the logger
     * and clears the buffer.
     */
    protected void log() {
        if (chopTerminatingNewLine) { chomp(); }
        logger.log(priority, buffer);
        buffer.delete(0,buffer.length());
    }

    /** chops the terminating new lines chars in the buffer, if there
     * are any.
     */
    protected final void chomp() {
        int length = buffer.length();
        switch (length) {
          case 0: break;
          case 1: {
            char last = buffer.charAt(0);
            if ((last == '\r') || (last == '\n')) {
                buffer.deleteCharAt(0);
            }
            break;
          }
          default: {
            char last = buffer.charAt(length-1);
            if (last == '\r') {
                buffer.deleteCharAt(length-1);
            } else if (last == '\n') {
                buffer.deleteCharAt(length-1);
                if (buffer.charAt(length-2) == '\r') {
                    buffer.deleteCharAt(length-2);
                }
            }
            //fall-out of the switch block
          }
        }
    }
}
```

Setting The Conversion Pattern Layout Differently Per Logging Level

This class was written to allow one to set the log output differently depending on the level of the log event. For example, you might want to output more verbose info for DEBUG log messages than for INFO. Thank the power of inheritance for my ease in implementing this!

```
package com.coolbeans.log4j;

import org.apache.log4j.PatternLayout;
import org.apache.log4j.spi.LoggingEvent;
import org.apache.log4j.Level;


/**
 * Extends the {@link org.apache.log4j.PatternLayout} class by adding the capability
 * to set different conversion patterns for each logging level.  This class will allow log entries
 * to be formatted differently according to the level of the log entry.  For example, entries at
 * the DEBUG level can contain more verbose information, such as the caller which generated the
 * logging event, while entries at the INFO level can be formatted with only a date and the message.
 * This class is thread safe to ensure the string returned by the {@link #format(LoggingEvent) format}
 * method is the one intended for the calling thread.
 *
```

```
 * @author Oliver C. Hernandez
 * @see org.apache.log4j.PatternLayout
 */
public class PatternLayoutByLevel extends PatternLayout {

  /** Default pattern string for logging at levels that do not have a pattern set. */
  protected String defaultPattern = DEFAULT_CONVERSION_PATTERN;

  /** Pattern string for logging at the DEBUG level. */
  protected String debugPattern   = DEFAULT_CONVERSION_PATTERN;

  /** Pattern string for logging at the INFO level. */
  protected String infoPattern    = DEFAULT_CONVERSION_PATTERN;

  /** Pattern string for logging at the WARN level. */
  protected String warnPattern    = DEFAULT_CONVERSION_PATTERN;

  /** Pattern string for logging at the ERROR level. */
  protected String errorPattern   = DEFAULT_CONVERSION_PATTERN;

  /** Pattern string for logging at the FATAL level. */
  protected String fatalPattern   = DEFAULT_CONVERSION_PATTERN;

  /** Indicator for when the pattern is set for the DEBUG level. */
  protected boolean debugPatternSet = false;

  /** Indicator for when the pattern is set for the INFO level. */
  protected boolean infoPatternSet  = false;

  /** Indicator for when the pattern is set for the WARN level. */
  protected boolean warnPatternSet  = false;

  /** Indicator for when the pattern is set for the ERROR level. */
  protected boolean errorPatternSet = false;

  /** Indicator for when the pattern is set for the FATAL level. */
  protected boolean fatalPatternSet = false;

  /**
   * Constructs a PatternLayoutByLevel using the DEFAULT_LAYOUT_PATTERN of the
   * superclass {@link org.apache.log4j.PatternLayout} for all log levels.
   */
  public PatternLayoutByLevel() {
    super();
  }

  /**
   * Constructs a PatternLayoutByLevel using the supplied conversion pattern
   * as the default pattern for log levels that do not have a pattern set.
   *
   * @param pattern the default pattern to format log events.
   */
  public PatternLayoutByLevel(String pattern) {
    super(pattern);
    this.defaultPattern = pattern;
  }

  /**
   * Set the <b>ConversionPattern</b> option.  This is the string which controls formatting and
   * consists of a mix of literal content and conversion specifiers.  This will be the
   * pattern for log levels that do not have a pattern set for them.
   *
   * @param conversionPattern pattern string to set to.
   */
  public void setConversionPattern(String conversionPattern) {
    this.defaultPattern = conversionPattern;
    super.setConversionPattern(conversionPattern);
  }

  /**
   * Set the <b>ConversionPattern</b> option for logging at the DEBUG level.
```

```
   *
   * @param pattern pattern string for logging at the DEBUG level.
   */
  public void setDebugPattern(String pattern) {
    this.debugPattern = pattern;
    this.debugPatternSet = true;
  }

  /**
   * Set the <b>ConversionPattern</b> option for logging at the INFO level.
   *
   * @param pattern pattern string for logging at the INFO level.
   */
  public void setInfoPattern(String pattern) {
    this.infoPattern = pattern;
    this.infoPatternSet = true;
  }

  /**
   * Set the <b>ConversionPattern</b> option for logging at the WARN level.
   *
   * @param pattern pattern string for logging at the WARN level.
   */
  public void setWarnPattern(String pattern) {
    this.warnPattern = pattern;
    this.warnPatternSet = true;
  }

  /**
   * Set the <b>ConversionPattern</b> option for logging at the ERROR level.
   *
   * @param pattern pattern string for logging at the ERROR level.
   */
  public void setErrorPattern(String pattern) {
    this.errorPattern = pattern;
    this.errorPatternSet = true;
  }

  /**
   * Set the <b>ConversionPattern</b> option for logging at the FATAL level.
   *
   * @param pattern pattern string for logging at the FATAL level.
   */
  public void setFatalPattern(String pattern) {
    this.fatalPattern = pattern;
    this.fatalPatternSet = true;
  }

  /**
   * Produces a formatted string according to the conversion pattern set for the level of the logging event
passed in.
   *
   * @param event log event to format an entry for.
   * @return a formatted log entry.
   */
  public synchronized String format(LoggingEvent event) {

    // Reset the conversion pattern in case it is not set for any log levels.
    super.setConversionPattern(this.defaultPattern);

    /*
     * Set the conversion pattern to format according to the log level of the event.
     * For each log level, check first if a pattern has been set for it.  This will
     * save execution time by not needlessly calling the methods to check the level
     * of the event.  If the pattern has been set for the log level, then check if
     * it matches the level of the event.  When both of these conditions are true,
     * the conversion pattern is set accordingly, and a format is returned by the
     * format method of the superclass.
     */

    if (this.debugPatternSet) {
```

```
      if (event.getLevel().equals(Level.DEBUG)) {
        super.setConversionPattern(this.debugPattern);
      }
    }

    if (this.infoPatternSet) {
      if (event.getLevel().equals(Level.INFO)) {
        super.setConversionPattern(this.infoPattern);
      }
    }

    if (this.warnPatternSet) {
      if (event.getLevel().equals(Level.WARN)) {
        super.setConversionPattern(this.warnPattern);
      }
    }

    if (this.errorPatternSet) {
      if (event.getLevel().equals(Level.ERROR)) {
        super.setConversionPattern(this.errorPattern);
      }
    }

    if (this.fatalPatternSet) {
      if (event.getLevel().equals(Level.FATAL)) {
        super.setConversionPattern(this.fatalPattern);
      }
    }

    return super.format(event);
  }


}
```

An example configuration:

```
log4j.rootLogger=DEBUG, defaultRoot

log4j.appender.defaultRoot=org.apache.log4j.DailyRollingFileAppender
log4j.appender.defaultRoot.DatePattern='.'yyyy-MM-dd
log4j.appender.defaultRoot.File=myapp.log

# Use the custom layout class
log4j.appender.defaultRoot.layout=com.coolbeans.log4j.PatternLayoutByLevel

# Set default format for log levels not specifically set
log4j.appender.defaultRoot.layout.ConversionPattern = %d{ISO8601} %-5p [%t] %m%n

# Set formats for different log levels
#log4j.appender.defaultRoot.layout.InfoPattern = %d{ISO8601} %-5p [%t] %m%n
log4j.appender.defaultRoot.layout.DebugPattern = %n%d{ISO8601} %-5p [%t]%n %r ms since application start%n %l%n
%m%n%n
log4j.appender.defaultRoot.layout.ErrorPattern = %n%d{ISO8601} %-5p [%t]%n %r ms since application start%n %l%n
%m%n%n
log4j.appender.defaultRoot.layout.FatalPattern = %n%d{ISO8601} %-5p [%t]%n %r ms since application start%n %l%n
%m%n%n
```

by Oliver Hernandez (www.diamonddata.com)

Having ConversionPattern based based on log level is very useful, but implementation above is rather inefficient, as every call to "format" create new PatternConverter, unlike it is being created only when log4j.properties is parsed.

As "sbuf" is private in PatternLayout and I need to use "my" PatternConverter, I had to create my own "sbuf" for those cases. Would be nice to have it protected.

I propose a bit optimized solution here if anybody cares to try:

```
public class PatternLayoutByLevelWithHeader extends PatternLayout {

        private Map<Level, PatternConverter> logLevelConverterMap = new HashMap<Level, PatternConverter>();
        private StringBuffer sbuf = new StringBuffer(BUF_SIZE);

        public PatternLayoutByLevelWithHeader() {
                super();
        }

        public PatternLayoutByLevelWithHeader(String conversionPattern) {
                super(conversionPattern);
        }

          /**
           * Set the <b>ConversionPattern</b> option.  This is the string which controls formatting and
           * consists of a mix of literal content and conversion specifiers.  This will be the
           * pattern for log levels that do not have a pattern set for them.
           *
           * @param conversionPattern pattern string to set to.
           */
          public void setConversionPattern(String conversionPattern) {
            super.setConversionPattern(conversionPattern);
          }

          /**
           * Set the <b>ConversionPattern</b> option for logging at the DEBUG level.
           *
           * @param pattern pattern string for logging at the DEBUG level.
           */
          public void setDebugPattern(String pattern) {
            logLevelConverterMap.put(Level.DEBUG, createPatternParser(pattern).parse());
          }

          /**
           * Set the <b>ConversionPattern</b> option for logging at the INFO level.
           *
           * @param pattern pattern string for logging at the INFO level.
           */
          public void setInfoPattern(String pattern) {
            logLevelConverterMap.put(Level.INFO, createPatternParser(pattern).parse());
          }

          /**
           * Set the <b>ConversionPattern</b> option for logging at the WARN level.
           *
           * @param pattern pattern string for logging at the WARN level.
           */
          public void setWarnPattern(String pattern) {
            logLevelConverterMap.put(Level.WARN, createPatternParser(pattern).parse());
          }

          /**
           * Set the <b>ConversionPattern</b> option for logging at the ERROR level.
           *
           * @param pattern pattern string for logging at the ERROR level.
           */
          public void setErrorPattern(String pattern) {
            logLevelConverterMap.put(Level.ERROR, createPatternParser(pattern).parse());
          }

          /**
           * Set the <b>ConversionPattern</b> option for logging at the FATAL level.
           *
           * @param pattern pattern string for logging at the FATAL level.
           */
          public void setFatalPattern(String pattern) {
            logLevelConverterMap.put(Level.FATAL, createPatternParser(pattern).parse());
          }

          /**
           * Produces a formatted string according to the conversion pattern set for the level of the logging
```

```
event passed in.
           *
           * @param event log event to format an entry for.
           * @return a formatted log entry.
           */
          public String format(LoggingEvent event) {
                    if ( logLevelConverterMap.containsKey(event.getLevel()) ) {
                            // Reset working stringbuffer
                            if(sbuf.capacity() > MAX_CAPACITY) {
                              sbuf = new StringBuffer(BUF_SIZE);
                            } else {
                              sbuf.setLength(0);
                            }
                            PatternConverter c = logLevelConverterMap.get(event.getLevel());


                            while(c != null) {
                              c.format(sbuf, event);
                              c = c.next;
                            }
                            return sbuf.toString();
                        } else {
                                return super.format(event);
                        }
          }
```

by Yuliya Feldman

Non-verbose layout formatter

I have created a new Layout that supports 2 new features:

- %P, that shows the 1st letter of the priority: D, I, W, E, F instead of Debug, Info, Warning, Error, Fatal. It is more compact and easy to understand.
- %T, for filtering stack traces, e.g. removing stack traces in web server code and other uninteresting places.

In this way the logs aren't so verbose.

Here is the source code:

LevelPatternParser.java LevelPatternLayout.java

by David Pérez (craquerpro at yahoo dot es)

HttpServlet XML initialization Example

```
package logging;

import ...;

public class LogInit extends HttpServlet
{

    static boolean isInit = false;

    public void init() {

        if( !isInit ){
            try{
                String file = getInitParameter("log-init-file");
                InputStream log4JConfig = getServletConfig().getServletContext()
                    .getResourceAsStream(file);

                Document doc = DocumentBuilderFactory.newInstance()
                    .newDocumentBuilder()
                    .parse(log4JConfig);

                DOMConfigurator.configure( doc.getDocumentElement() );
                isInit = true;
            }
            catch( ParserConfigurationException pce){
                System.out.println( "*** Failed to initialize log4j configuration ***" );
                //Do something specific here if required.
                pce.printStackTrace();
            }
            catch( SAXException se){
                System.out.println( "*** Failed to initialize log4j configuration ***" );
                //Do something specific here if required.
                se.printStackTrace();
            }
            catch( IOException ioe){
                System.out.println( "*** Failed to initialize log4j configuration ***" );
                //Do something specific here if required.
                ioe.printStackTrace();
            }
        }
    }
}
```

web.xml contains

```
<servlet>
    <servlet-name>log-init</servlet-name>
    <servlet-class>logging.LogInit</servlet-class>
    <init-param>
      <param-name>log-init-file</param-name>
      <param-value>WEB-INF/classes/log4j.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

See the documentation included in the download for XML formatting examples.

by David Webster

JDK 1.3 Service Definition Configurator

I have created a custom configurator which uses a JDK 1.3 service definition to discover the configurator class at runtime.

```
package com.hca.common.logging;

import java.net.URL;
import java.util.Iterator;

import org.apache.log4j.helpers.OptionConverter;
import org.apache.log4j.spi.Configurator;
import org.apache.log4j.spi.LoggerRepository;

import sun.misc.Service;

/**
 * A Log4jConfigurator which enables custom log4j configurators to be installed as
 * JDK 1.3 style services (/META-INF/services/org.apache.log4j.spi.Configurator).
 *
 * @author Chris Lance
 */
public class JDKServiceConfigurator
        implements Configurator
{
        /**
         * Looks up the service definition and then calls it's <code>doConfigure</code> method
         * with the supplied url and repository.
         *
         * @see org.apache.log4j.spi.Configurator#doConfigure(java.net.URL, org.apache.log4j.spi.
LoggerRepository)
         */
        public void doConfigure(URL url, LoggerRepository repo)
        {
                Iterator itr = Service.providers(Configurator.class);
                if(itr.hasNext())
                {
                        Configurator cfg = (Configurator) itr.next();
                        cfg.doConfigure(url, repo);
                }
                else
                {
                        //Do default configuration procedure
                        OptionConverter.selectAndConfigure(url, null, repo);
                }
        }
}
```

Set the system property *log4j.configuratorClass = com.hca.common.logging.JDKServiceConfigurator*. In your WAR, create the file *META-INF/services/org.apache.log4j.spi.Configurator* and set its contents to the name of your custom configurator class. Works great for servers running multiple applications which each need custom configuration.

This version depends on JDK 1.3+ API's, but it can easily be modified to use commons discovery or a custom implementation.

by Chris Lance

Timestamp File Appender

Appender creates a new log file on each start of an application, with the additional feature that the log file name contains the current timestamp.

Appender is pretty simple, it is derived from *FileAppender* and replaces string *{timestamp}* from the value of *File* property with the current timestamp.

Timestamp pattern is user configured by *TimestampPattern* property that has the same choice of formats as *java.text.SimpleDateFormat*

So, the solution is very flexible.

Here are the source code and configuration example file:

TimestampFileAppender.java timestampFileAppender.configuration

by Viktor Bresan

Each Application Launch File Appender

This Appender creates a new log file on each start of an application (like the *Timestamp File Appender*). But the previous log file is backed up as *logfilenam e.n* where *n* is the nth previous launch, up to a specified limit.

The Appender is pretty simple, it is derived from *FileAppender* and checks for previous logs, shiffting the log number down. The modification date of the logs files are preserved.

Here's the source code:

```
import java.io.*;
import java.util.*;

import org.apache.log4j.*;
import org.apache.log4j.helpers.*;
import org.apache.log4j.spi.*;

/**
 * Provides a log file per execution.
 * The processing is done during initilaisation.
 * We check if a log file of the same name already exists, if so, we back the previous one as "logfilename.1".
 * We can keep track of several back ups (see the maxLogArchives property), by shiffting their number
 * (ie if "logfilename.1" exists, rename to "logfilename.2" and "logfilename" becomes "logfilename.1" and we log
 * to "logfilename" ).
 *
 * cf http://news.gmane.org/find-root.php?message_id=%3cc83e39890610250805l1ca82c4v3cbbd223b83f13d8%40mail.
gmail.com%3e
 *
 * <pre>
 * log4j.rootLogger=DEBUG, exeroll
 * log4j.appender.exeroll=EachLaunchFileAppender
 * log4j.appender.exeroll.File=./log/client.log
 * # optional, default is 5
 * log4j.appender.exeroll.maxLogArchives=10
 * log4j.appender.exeroll.layout=org.apache.log4j.PatternLayout
 * log4j.appender.exeroll.layout.ConversionPattern=%d{ABSOLUTE} [%t] [%-5p] [%-25.48c{1}] - %m - [%l] %n
 * </pre>
 *
 * This is just a quick hack, it works for me, do whatever you want with it
 * Date: 25 oct. 2006
 *
 * @author Chris Dillon
 */
public class EachLaunchFileAppender extends FileAppender {

        private int maxLogArchives = 5;

        public EachLaunchFileAppender() {
        }

        public EachLaunchFileAppender(Layout pLayout, String filename) throws IOException {
                super(pLayout, filename);
        }

        public EachLaunchFileAppender(Layout pLayout, String filename, boolean append) throws IOException {
                super(pLayout, filename, append);
        }

        public EachLaunchFileAppender(Layout pLayout, String filename, boolean append, boolean pBufferedIO,
                                                                        int pBufferSize) throws IOException {
                super(pLayout, filename, append, pBufferedIO, pBufferSize);
        }

        public void activateOptions() {
                if (fileName != null) {
                        try {
                                handleBackup();
                                // Super processing
                                setFile(fileName, fileAppend, bufferedIO, bufferSize);
                        }
                        catch (IOException e) {
                                errorHandler.error("setFile(" + fileName + "," + fileAppend + ") call failed.",
                                                e, ErrorCode.FILE_OPEN_FAILURE);
```

```
                        }
                }
                else {
                        LogLog.warn("File option not set for appender [" + name + "].");
                        LogLog.warn("Are you using FileAppender instead of ConsoleAppender?");
                }
        }

        /**
         * If the file (fileName) to log to already exists, we try to back it up to fileName.1.
         * <br>
         * If previous backups already exist, we rename each of them to n+1 (ie: log.1 becomes log.2 etc...)
         * until we reache the value set by {@link #maxLogArchives} of log archives (we delete the oldest).
         */
        private void handleBackup() {
                final File newLogFile = new File(fileName);
                if (!newLogFile.exists()) {
                        return;
                }
                final File dir = newLogFile.getParentFile();
                final String logFileName = newLogFile.getName();
                String[] oldLogNames = dir.list(new FilenameFilter() {
                        String dotName = logFileName + ".";
                        public boolean accept(File pDir, String pName) {
                                return pName.startsWith(dotName);
                        }
                });
                // go through names from older to younger
                Arrays.sort(oldLogNames,new ArchiveComparator());
                for (int i = oldLogNames.length -1; i >= 0; i--) {
                        final String oldLogName = oldLogNames[i];
                        try {
                                int fileNo = getArchiveNumber(oldLogName);
                                if (fileNo >= maxLogArchives){
                                        LogLog.debug("deleting : "+oldLogName);
                                        File del = new File(dir,oldLogName);
                                        del.delete();
                                }
                                else{
                                        fileNo++;
                                        File toBackup = new File(dir, oldLogName);
                                        final long lastModif = toBackup.lastModified();
                                        final File backupTo = new File(fileName + "." + fileNo);
                                        LogLog.debug("Renamming " + toBackup + " to" + backupTo);
                                        boolean renOk = toBackup.renameTo(backupTo);
                                        backupTo.setLastModified(lastModif);
                                        LogLog.debug("Renamming " + toBackup + " to" + backupTo + " done :" +
renOk);
                                }
                        }
                        catch (Throwable e) {
                                LogLog.warn("Error during back up of "+oldLogName,e);
                        }
                }
                final long lastModif = newLogFile.lastModified();
                final File backupTo = new File(fileName + ".1");
                LogLog.debug("Renamming " + newLogFile + " to" + backupTo);
                boolean renOk = newLogFile.renameTo(backupTo);
                LogLog.debug("Renamming " + newLogFile + " to" + backupTo + " done :" + renOk);
                backupTo.setLastModified(lastModif);
        }

        /**
         * returns the number of the archive.
         * log.2 return 2
         * @param pFileName
         * @return
         */
        private int getArchiveNumber(String pFileName){
                int lastDotIdx = pFileName.lastIndexOf('.');
                String oldLogNumber = pFileName.substring(lastDotIdx + 1);
```

```
                return Integer.parseInt(oldLogNumber);
        }

        private class ArchiveComparator implements Comparator<String> {
                public int compare(String p1, String p2) {
                        final int one = getArchiveNumber(p1);
                        final int two = getArchiveNumber(p2);
                        return one - two;
                }
        }



        public int getMaxLogArchives() {
                return maxLogArchives;
        }

        public void setMaxLogArchives(int pMaxLogArchives) {
                maxLogArchives = pMaxLogArchives;
        }

}
```

by Chris Dillon

Configuring Log4J For Testing With Maven The following posting appeared on the Log4J Users mailing list:

I use Log4J and build my applications with Maven2. I have a log4j.xml config file in the src/main/resources directory so that Maven2 builds it into the JAR with my application.

However, I have a different log4j.xml that I want to use with the JUnit test that run as part of my Maven build process. I'd like to put that log4j.xml (testing) in the src/test/resources directory and have it overide the production version in src/main/resources. However, Maven doesn't seem to work that way. Instead, the production version of log4j.xml seems to come first on the testing classpath and therefore gets loaded instead of the testing version of log4j.xml.

I'm sure others must have faced this issue. Are there any recommended solutions to the problem?

Here is the solution that was presented:

1. Rename the log4j.xml file that you use for testing to something else, perhaps test-log4j.xml. Keep it in the src/test/resources directory.

2. Make sure that you've included a testResource entry in your pom that includes the test-log4j.xml file. Something like this:

```
<build>
  <testResources>
    <testResource>
      <directory>src/test/resources</directory>
      <includes>
        <include>test-log4j.xml</include>
      </includes>
    </testResource>
  </testResources>
</build>
```

This entry will force Maven to copy the test-log4j.xml file from the src/test/resources directory to the target/test-classes whenever it compiles the test classes.

3. Tell log4j to use your test configuration file during the unit tests. This is done by configuring the surefire plugin. More specifically, you'll set the "log4j.configuration" system property to "test-log4j.xml". Details on configuring the surefire plugin can be found here. Here is what I believe you'll need to configure the surefire plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <systemProperties>
          <property>
            <name>log4j.configuration</name>
            <value>test-log4j.xml</value>
          </property>
        </systemProperties>
      </configuration>
    </plugin>
  </plugins>
</build>
```

With these changes in place, the following will happen:

1. When your test classes are compiled, Maven will copy the test-log4j.xml file from ${basedir}/src/test/resources to ${basedir}/target/test-classes.

2. When you run your tests, Maven will include the ${basedir}/target/test-classes directory in the class path.

3. Since you've set the "log4j.configuration" system property to "test-log4j.xml", log4j will configure itself using the contents of that file, rather than the default "log4j.xml" file.

by Ron Gallagher