**FACULTAD DE INGENIERIA DE SISTEMAS E INFORMÁTICA**

**UCH**

**Integrantes: Rivera Quispe, David**

**Carrión Del Pino, Daniel**

**Tema: Log (Registro)**

**Profesor: Gustavo Coronel**

**Ciclo: III**

**Curso: Programación I (Java POO)**

**Año: 2016**

**Índice**

**RESUMEN Pág. 3**

**ANTECEDENTES Pág. 4**

**DESARROLLO DEL TEMA Pág. 5**

**CASOS Pág. 11**

**CONCLUSIONES Pág. 20**

**RECOMENDACIONES Pág. 21**

**BIBLIOGRAFÍA Pág. 22**

**ANEXOS Pág. 23**

**Resumen**

En este tema vamos a tratar sobre que es un log y para que se utiliza:

Un log es un registro de actividad de un sistema, que generalmente se guarda en un fichero de texto, al que se le van añadiendo líneas a medida que se realizan acciones sobre el sistema.

Se utiliza en muchos casos distintos, para guardar información sobre la actividad de sistemas variados.

Tal vez su uso más inmediato a nuestras actividades como desarrolladores del web sería el log de accesos al servidor web, que analizado da información del tráfico de nuestro sitio. Cualquier servidor web dispone de un log con los accesos, pero, además, suelen disponer de otros log, por ejemplo, de errores.

Los sistemas operativos también suelen trabajar con logs, por ejemplo, para guardar incidencias, errores, accesos de usuarios, etc.

A través de los log se puede encontrar información para detectar posibles problemas en caso de que no funcione algún sistema como debiera o se haya producido una incidencia de seguridad.

**Antecedentes**

Nosotros hemos escogido el tema de log ya que nos gustó el procesamiento exacto de todos nuestros métodos en las aplicaciones que realizamos en Java en el curso de programación POO.

Para ver minimizar los errores que mayormente se comenten en la programación de las aplicaciones y en este caso nos llamó la atención porque mediante los conceptos de log podemos analizar este caso y solucionarlo de la manera correcta y ordenada.

**Desarrollo del Tema**

En este tema mostraremos **como utilizar logs de Java** para llevar un registro de los sucesos importantes o de depuración de una aplicación Java. Con la finalidad de generar archivos de eventos para futuros diagnósticos de la aplicación.

El control de mensajes será incluido en nuestra aplicación como código duro y será definido por nosotros, lo que podemos dejar para ser administrado dinámicamente es a que nivel de detalle queremos reportar. Entonces yendo al punto, una forma común de incluir mensajes de log dentro de nuestra aplicación es de la siguiente forma:

import java.util.logging.Level;

import java.util.logging.Logger;

...

public void metodo() {

try {

...

Logger.getLogger(getClass().getName()).log(

Level.INFO, "Mensaje informativo...");

...

} catch (Exception err) {

...

Logger.getLogger(getClass().getName()).log(

Level.SEVERE, "Mensaje crítico...", err);

...

}

}

Existen diferentes niveles (Level.[…]) para la cual desplegar un mensaje, los niveles son:

* **INFO:** Mostrar mensajes informativos
* **CONFIG:** Mostrar mensajes de configuración
* **WARNING:** Mostrar mensajes de alerta
* **SEVERE:** Mostrar mensajes críticos
* **FINE:** Mostrar mensajes de depuración de nivel 1
* **FINER:** Mostrar mensajes de depuración de nivel 2
* **FINEST:** Mostrar mensajes de depuración de nivel 3

Para administrar el nivel de detalle de log que queremos reportar a consola o a un archivo, se utilizan los mismos niveles definidos anteriormente (INFO, CONFIG, etc.) y además también pueden utilizarse:

* **ALL:** Muestra todos los mensajes
* **OFF:** No muestra ningún mensaje

Una vez que la aplicación ya contiene las líneas de código para mostrar mensajes, podemos **administrar la salida de estos mensajes**, es decir, si queremos que vayan a un archivo, a pantalla ó a un socket, etc…, existen dos maneras de llevar acabo esta configuración: **usando archivos de configuración** o **realizando la configuración desde el código de la aplicación**, vamos a mostrar las dos formas pero preferimos realizar esta administración a través de archivos de configuración ya que permiten cambiar el nivel de log sin modificar el código fuente.

**Usando archivos de configuración**

Primero debemos **especificar donde se encuentra nuestro archivo de configuración de log**. También existen dos formas para indicar dónde está el archivo, la primera desde la ejecución de la aplicación con el parámetro *-Djava.util.logging.config.file*:

java -Djava.util.logging.config.file=./log.properties

-cp ./classes:./lib/\* ClassMain

Y la segunda desde código fuente:

public void metodo() {

LogManager.getLogManager().readConfiguration(

new FileInputStream("./log.properties"));

}

Un ejemplo de archivo de propiedades de log (“log.properties”) es:

# especificacion de detalle de log

# nivel de log global

.level = WARNING

# nivel de log unicamente para la clase pkg.subpkg.JobClass

pkg.subpkg.JobClass.level = INFO

# manejadores de salida de log

# se cargaron un manejador de archivos y

# manejador de consola

handlers = java.util.logging.FileHandler,

java.util.logging.ConsoleHandler

# configuración de manejador de archivos

# nivel soportado para archivos

java.util.logging.FileHandler.level = ALL

# archivo de almacenamiento de las salidas de log

java.util.logging.FileHandler.pattern = ./log/app%g.log

# maximo tamaño de archivo en bytes

java.util.logging.FileHandler.limit = 10485760

# maximo numero de archivos de logs

java.util.logging.FileHandler.count = 3

# clase para formatear salida hacia el archivo de log

java.util.logging.FileHandler.formatter =

java.util.logging.XMLFormatter

# anexar entrada al último archivo (si es falso escribirá al

# inicio del archivo cuando la aplicación sea ejecutada)

java.util.logging.FileHandler.append = true

# configuración de manejador de consola

# nivel soportado para consola

java.util.logging.ConsoleHandler.level = SEVERE

# clase para formatear salida hacia consola

java.util.logging.ConsoleHandler.formatter =

java.util.logging.SimpleFormatter

Como consejo, a menudo el despliegue a consola es controlado por nosotros para reportar mensajes más fáciles de entender al usuario, así que para eliminar los logs a consola utiliza*java.util.logging.ConsoleHandler.level = OFF*

Con esto podemos dar por terminado el uso de logs en Java.

**Configuración de logs desde código fuente**

Para tener un parámetro de referencia realizaré la misma configuración que se tiene utilizando el archivo de configuración, pero ahora desde código fuente.

import java.util.logging.\*;

...

public void metodo()

throws SecurityException, IOException {

// creando manejador de archivo

FileHandler fh = new FileHandler(

"./log/app%g.log", //pattern

10485760, //limit

3, // count

true); //append

fh.setLevel(Level.ALL); // level

fh.setFormatter(new XMLFormatter()); //formatter

// agregar el manejador de archivo al log

Logger.global.addHandler(fh);

// el manejador de consola se agrega automaticamente, solo

// cambiamos el nivel de detalle a desplegar

Logger.global.getHandlers()[0].setLevel(Level.SEVERE);

// se establece el nivel predeterminado global

Logger.global.setLevel(Level.INFO);

}

...

Bueno como ya vimos en la anterior entrada [(parte 1)](https://danjared.wordpress.com/2009/05/11/usar-logs-en-java-parte-1/) pudimos generar una salida de log muy general, pero ahora veremos cómo personalizar el funcionamiento de los logs en las siguientes secciones:

* Filtros
* Formatos
* Manejadores

**Filtros**

Primero veamos los filtros, para entender este punto planteare la siguiente necesidad: *reportar únicamente mensajes de depuración (FINE, FINER, FINEST), alarmas (WRNING) y críticos (SEVERE), sin reportar informativos (INFO) y de configuración (CONFIG)*. Si utilizamos el nivel ALL me mostrará todos los mensajes incluyendo INFO y CONFIG así que será necesario crear una clase de filtro:

package jared.util.logging;

import java.util.logging.Filter;

import java.util.logging.Level;

import java.util.logging.LogRecord;

public class FilterPersonalizado implements Filter {

public boolean isLoggable(LogRecord record) {

if (record.getLevel().equals(Level.FINE) ||

record.getLevel().equals(Level.FINER) ||

record.getLevel().equals(Level.FINEST) ||

record.getLevel().equals(Level.WARNING) ||

record.getLevel().equals(Level.SEVERE)) {

return true;

} else {

return false;

}

}

}

Ahora para poder utilizar el filtro lo mostraré de dos formas

*Usando archivo de configuración (log.properties)*

...

java.util.logging.FileHandler.filter =

jared.util.logging.FilterPersonalizado

...

Desde código fuente

...

public void metodo()

throws SecurityException, IOException {

// creando manejador de archivo

FileHandler fh = new FileHandler(

"./log/app%g.log", //pattern

10485760, //limit

3, // count

true); //append

fh.setLevel(Level.ALL); // level

fh.setFilter(new FilterPersonalizado()); // filtro

fh.setFormatter(new FormatterPersonalizado()); // formato

// agregar el manejador de archivo al log

Logger.global.addHandler(fh);

// el manejador de consola se agrega automaticamente, solo

// cambiamos el nivel de detalle a desplegar

Logger.global.getHandlers()[0].setLevel(Level.SEVERE);

// se establece el nivel predeterminado global

Logger.global.setLevel(Level.INFO);

}

...

**Manejadores**

Realmente en manejadores de log Java es muy completo y posee los siguientes manejadores:

* java.util.logging.ConsoleHandler: Muestra mensajes a consola.
* java.util.logging.FileHandler: Guarda los mensajes a archivos.
* java.util.logging.MemoryHandler: Guarda los mensajes en memoría.
* java.util.logging.SocketHandler: Envia los mensajes a un socket.
* java.util.logging.StreamHandler: Envia los mensajes a un flujo de bytes.

Ya hemos visto cómo utilizar ConsoleHandler y FileHandler, para está entrada solo mostraré como utilizar SocketHandler por considerar que puede ser el más usado después de ConsoleHandler y FileHandler.

Mostraré como utilizarlo también de las dos formas:

Usando archivo de configuración (log.properties)

...

# nivel soportado para socket

java.util.logging.SocketHandler.level = ALL

# filtro de mensajes

java.util.logging.SocketHandler.filter =

jared.util.logging.FilterPersonalizado

# formato de salida

java.util.logging.SocketHandler.formatter =

jared.util.logging.FormatterPersonalizado

# host donde se recibirá la información

java.util.logging.SocketHandler.host = ubuntu-laptop

# puerto donde se recibirá la información

java.util.logging.SocketHandler.port = 1000

...

**Desde código fuente**

...

import java.io.IOException;

import java.util.logging.Level;

import java.util.logging.Logger;

import java.util.logging.SocketHandler;

...

public void metodo() throws IOException {

SocketHandler sh = new SocketHandler(

"ubuntu-laptop", 1000);

sh.setFilter(null);

sh.setFormatter(null);

sh.setLevel(Level.ALL);

Logger.global.addHandler(sh);

}

...

Con esto ya estaría, ahora necesitamos el concentrador de logs en red que no es más que un ServerSocket común. Les dejo un ejemplo de cómo recibir los mensajes desde un SocketHandler que imprime la salida a pantalla.

...

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.ServerSocket;

import java.net.Socket;

...

public void metodo() throws IOException {

ServerSocket serverLog = new ServerSocket(1000);

while (true) {

final Socket socket = serverLog.accept();

new Thread() {

public void run() {

try {

BufferedReader buffer = new BufferedReader(

new InputStreamReader(

socket.getInputStream()));

String line = "";

while ((line = buffer.readLine()) != null)

System.out.println(line);

buffer.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}.start();

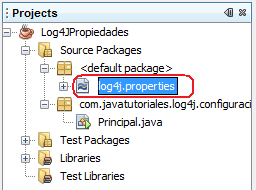
}

}

...

**Casos**

Si queremos configurar log4j haciendo uso de un archivo de propiedades, lo primero que debemos hacer es crear un archivo de propiedades (properties file) en el paquete default de la aplicación. Este archivo debe llamarse "log4j.properties" (podríamos llamar este archivo de otra forma o colocarlo en otro lugar, pero log4j busca por default este archivo en esta ubicación, haciéndolo de esta forma nos evitamos el tener realizar una configuración de forma programática en nuestra aplicación):



Lo primero que hay que hacer es crear un appender, al cual le daremos un nombre arbitrario, e indicaremos el tipo de appender que será, usando una de las clases que mencionamos hace un momento, cuando hablamos de los appenders. En este caso se creará un appender llamado "consola" y que será de tipo "org.apache.log4j.ConsoleAppender", que permite enviar los mensjes a la salida estándar, además indicaremos que los mensajes deberán mostrarse en "System.out", que es su valor por default (también podemos mandar los mensajes a "System.err"):

log4j.appender.consola = org.apache.log4j.ConsoleAppender

log4j.appender.consola.target = System.out

Lo que sigue es agregar un layout para este appender. Existen muchos tipos de layouts. Para esta primera prueba usaremos el "SimpleLayout", que simplemente nos indica el nivel de log del logger, y el mensaje asociado al evento:

log4j.appender.consola.layout = org.apache.log4j.SimpleLayout

Una vez que se ha creado el appender, y que se la ha establecido su layout, se debe especificar cuáles loggers usarán este appender, y establecer su nivel de log. Si se especifica que "rootLogger" hará uso de este appender, entonces todos los loggers enviarán sus mensajes a este appender (a menos que se especifique lo contrario de forma explícita). Como rootLogger está en la cima de la jerarquía, todos los loggers heredarán su nivel de log y su appender:

log4j.rootLogger = TRACE, consola

El archivo de configuración queda de la siguiente forma:

log4j.appender.consola = org.apache.log4j.ConsoleAppender

log4j.appender.consola.target = System.out

log4j.appender.consola.layout = org.apache.log4j.SimpleLayout

log4j.rootLogger = TRACE, consola

Al ejecutar el código anterior se obtiene la siguiente salida:

**TRACE - mensaje de trace**

**DEBUG - mensaje de debug**

**INFO - mensaje de info**

**WARN - mensaje de warn**

**ERROR - mensaje de error**

**FATAL - mensaje de fatal**

Ahora, teniendo este archivo de configuración, es posible modificar toda la configuración de la salida. Por ejemplo, modificamos el layout del appender "consola" para que en vez de ser de tipo "SimpleLayout" ahora sea de tipo "HTMLLayout":

log4j.appender.consola.layout = org.apache.log4j.HTMLLayout

Obtendremos la siguiente salida:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<title>Log4j Log Messages</title>

<style type="text/css">

<!--

body, table {font-family: arial,sans-serif; font-size: x-small;}

th {background: #336699; color: #FFFFFF; text-align: left;}

-->

</style>

</head>

<body bgcolor="#FFFFFF" topmargin="6" leftmargin="6">

<hr size="1" noshade>

Log session start time Fri Apr 22 01:59:55 CDT 2011<br>

<br>

<table cellspacing="0" cellpadding="4" border="1" bordercolor="#224466" width="100%">

<tr>

<th>Time</th>

<th>Thread</th>

<th>Level</th>

<th>Category</th>

<th>Message</th>

</tr>

<tr>

<td>0</td>

<td title="main thread">main</td>

<td title="Level">TRACE</td>

<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal</td>

<td title="Message">mensaje de trace</td>

</tr>

<tr>

<td>3</td>

<td title="main thread">main</td>

<td title="Level"><font color="#339933">DEBUG</font></td>

<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal</td>

<td title="Message">mensaje de debug</td>

</tr>

<tr>

<td>3</td>

<td title="main thread">main</td>

<td title="Level">INFO</td>

<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal</td>

<td title="Message">mensaje de info</td>

</tr>

<tr>

<td>17</td>

<td title="main thread">main</td>

<td title="Level"><font color="#993300"><strong>WARN</strong></font></td>

<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal</td>

<td title="Message">mensaje de warn</td>

</tr>

<tr>

<td>20</td>

<td title="main thread">main</td>

<td title="Level"><font color="#993300"><strong>ERROR</strong></font></td>

<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal</td>

<td title="Message">mensaje de error</td>

</tr>

<tr>

<td>25</td>

<td title="main thread">main</td>

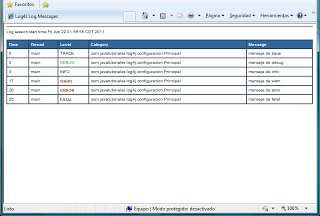
<td title="Level"><font color="#993300"><strong>FATAL</strong></font></td>

<td title="com.javatutoriales.log4j.configuracion.Principal category">com.javatutoriales.log4j.configuracion.Principal</td>

<td title="Message">mensaje de fatal</td>

</tr>

Que genera la siguiente página:



Como podemos ver, hemos modificado completamente la salida, sin necesidad de mover una sola línea de la aplicación (bueno, solo una línea de un archivo de propiedades, que siempre será un archivo de texto plano).

De la misma forma, si usamos un XMLLayout:

log4j.appender.consola.layout = org.apache.log4j.xml.XMLLayout

Obtendremos la siguiente salida:

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881257" level="TRACE" thread="main">

<log4j:message><![CDATA[mensaje de trace]]></log4j:message>

</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881259" level="DEBUG" thread="main">

<log4j:message><![CDATA[mensaje de debug]]></log4j:message>

</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881259" level="INFO" thread="main">

<log4j:message><![CDATA[mensaje de info]]></log4j:message>

</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881259" level="WARN" thread="main">

<log4j:message><![CDATA[mensaje de warn]]></log4j:message>

</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881262" level="ERROR" thread="main">

<log4j:message><![CDATA[mensaje de error]]></log4j:message>

</log4j:event>

<log4j:event logger="com.javatutoriales.log4j.configuracion.Principal" timestamp="1303485881262" level="FATAL" thread="main">

<log4j:message><![CDATA[mensaje de fatal]]></log4j:message>

</log4j:event>

Hagamos un último cambio de layout. Con los layout anteriores, la información que obtenemos podría no cumplir con nuestras expectativas, por ejemeplo, no hay forma de obtener información de qué método fue el que envió el mensaje, o usar un formato propio para la fecha, o agregar alguna información constante (alguna marca por ejemplo) en la salida generada. Para estos casos, en los que necesitamos hacer uso de un "EnhancedPatternLayout" (en versiones anteriores de log4j usábamos solo un "PatternLayout"). Este layout permite definir cómo será la salida de nuestros mensajes, usando una cadena con el patrón conocida como patrón de conversión.

El patrón de conversión es similar al de la función "printf", está compuesto de texto literal y expresiones de control de formato llamados especificadores de conversión. Cada especificador de conversión inicia con un símbolo de porcentaje (%) y es seguido de uno o más modificadores de formato y por un caracter de conversión. El caracter de conversión especifica el tipo de dato (por ejemplo, el logger, prioridad, fecha, etc.). Los modificadores de formato controlan cosas como el ancho del campo, justificación izquierda o derecha, etc. Al final del tutorial colocaré dos tablas, una con los caracteres de conversión, y otra con los modificadores de formato. Ahora veamos cómo usar este layout:

log4j.appender.consola.layout = org.apache.log4j.EnhancedPatternLayout

Para este layout debemos indicar su patrón de conversión, en su propiedad "ConversionPattern":

log4j.appender.consola.layout.ConversionPattern = [%-5p] %c{2} - %m%n

Con la configuración anterior se producirá la siguiente salida:

**[TRACE] configuracion.Principal - mensaje de trace**

**[DEBUG] configuracion.Principal - mensaje de debug**

**[INFO ] configuracion.Principal - mensaje de info**

**[WARN ] configuracion.Principal - mensaje de warn**

**[ERROR] configuracion.Principal - mensaje de error**

**[FATAL] configuracion.Principal - mensaje de fatal**

Podemos notar que ahora tenemos un poco más de información en la consola (como el nombre del logger). También noten que agregamos unas cuantas cadenas literales solo para que el la salida sea un poco más clara (como los signos "[" y "]" alrededor del nivel de log, y el "-" antes del mensaje).

Si a la salida anterior quisiéramos agregarle, por ejemplo, la fecha; bastaría con modificar el "ConversionPattern" de la siguiente forma:

log4j.appender.consola.layout.ConversionPattern = %d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n

Con lo que obtendríamos la siguiente salida:

22 abr 2011 - 10:47:47 [TRACE] configuracion.Principal - mensaje de trace

22 abr 2011 - 10:47:47 [DEBUG] configuracion.Principal - mensaje de debug

22 abr 2011 - 10:47:47 [INFO ] configuracion.Principal - mensaje de info

22 abr 2011 - 10:47:47 [WARN ] configuracion.Principal - mensaje de warn

22 abr 2011 - 10:47:47 [ERROR] configuracion.Principal - mensaje de error

22 abr 2011 - 10:47:47 [FATAL] configuracion.Principal - mensaje de fatal

De la misma forma que podemos modificar el layout de salida, podemos elegir que se muestren solo ciertos mensajes de log de un nivel determinado hacia arriba. Por ejemplo, para mostrar solo nos mensajes de nivel "WARN" a "FATAL" debemos configurar el logger de la siguiente forma:

log4j.rootLogger=WARN, consola

Con lo que los mensajes que obtenemos son los siguientes:

22 abr 2011 - 10:50:03 [WARN ] configuracion.Principal - mensaje de warn

22 abr 2011 - 10:50:03 [ERROR] configuracion.Principal - mensaje de error

22 abr 2011 - 10:50:03 [FATAL] configuracion.Principal - mensaje de fatal

Una última cosa importante que hay que saber, es que dentro del mismo appender podemos definir un umbral inferior para los mensajes que dicho appender mostrará. Por ejemplo, definimos que el appender "consola" tendrá un umbral inferior (threshold) para los mensajes de nivel "INFO", con la siguiente línea:

log4j.appender.consola.threshold = INFO

No importa que definamos el nivel del logger como TRACE:

log4j.rootLogger = TRACE, consola

Los únicos mensajes que se mostrarán para este appender son del nivel "INFO" hacia arriba:

22 abr 2011 - 10:54:46 [INFO ] configuracion.Principal - mensaje de info

22 abr 2011 - 10:54:46 [WARN ] configuracion.Principal - mensaje de warn

22 abr 2011 - 10:54:46 [ERROR] configuracion.Principal - mensaje de error

22 abr 2011 - 10:54:46 [FATAL] configuracion.Principal - mensaje de fatal

Lamentablemente en los archivos de propiedades solo podemos definir un nivel inferior, no uno superior (en los archivos de configuración XML es posible definir ambos niveles) haciendo uso de filtros.

Hasta ahora nuestro archivo de configuración se ve de la siguiente forma:

log4j.appender.consola = org.apache.log4j.ConsoleAppender

log4j.appender.consola.threshold = INFO

log4j.appender.consola.target = System.out

log4j.appender.consola.layout = org.apache.log4j.EnhancedPatternLayout

log4j.appender.consola.layout.ConversionPattern = %d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n

log4j.rootLogger = TRACE, consola

Como pudimos ver, con solo realizar una modificación al archivo de propiedades fue posible modificar el formato de los mensajes de salida, así como el nivel de log de la aplicación.

Es posible definir múltiples appenders en un mismo archivo de configuración, y asociar estos apennders a un mismo, o a diferentes loggers.

Por ejemplo, para crear un appender que envíe los mensajes de log a un archivo, sería necesario crear un FileAppender como el siguiente:

log4j.appender.archivo = org.apache.log4j.FileAppender

log4j.appender.archivo.file = archivo.log

log4j.appender.archivo.layout = org.apache.log4j.EnhancedPatternLayout

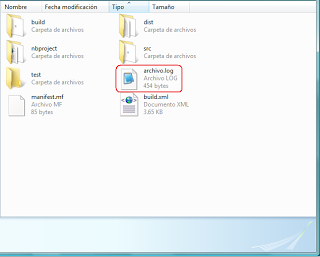
log4j.appender.archivo.layout.ConversionPattern = %d [%-5p] %c{2} - %m%n

Y para asociarlo al rootLogger, junto con el appender "consola" que ya se tenía configurado, se hace lo siguiente:

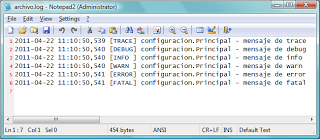
log4j.rootLogger = TRACE, consola, archivo

En este caso ambos appenders tienen el mismo nivel de log.

Cuando ejecutemos nuestra aplicación veremos aparecer un archivo llamado "archivo.log" en el directorio raíz de la aplicación:



Este archivo tiene el siguiente contenido:



Hay que tener cuidado de no cometer el error de tratar de que unos mensajes de log, con cierta prioridad, se envíen a un appender, y otros mensajes, con otra prioridad, se envíen a otro appender, de la siguiente forma:

log4j.rootLogger = TRACE, consola

log4j.rootLogger = ERROR, archivo

Con esto, lo único que se consigue es sobre-escribir el appender que estaba asociado al rootLogger, y que este deje de mostrar mensajes en la consola.

Es posible asociar appenders, no solo al rootLogger, sino también a loggers particulares, haciendo uso de su nombre. El appender que asociemos a un logger será heredado por todos sus descendientes. Por ejemplo, para asociar el appender "archivo" al logger "com.javatutoriales.log4j.configuracion", se hace lo siguiente:

log4j.logger.com.javatutoriales.log4j.configuracion = WARN, archivo

Con lo que el archivo de configuración quedaría de la siguiente forma:

log4j.appender.consola = org.apache.log4j.ConsoleAppender

log4j.appender.consola.threshold = INFO

log4j.appender.consola.target = System.out

log4j.appender.consola.layout = org.apache.log4j.EnhancedPatternLayout

log4j.appender.consola.layout.ConversionPattern = %d{dd MMM yyyy - HH:mm:ss} [%-5p] %c{2} - %m%n

log4j.appender.archivo = org.apache.log4j.FileAppender

log4j.appender.archivo.file = archivo.log

log4j.appender.archivo.layout = org.apache.log4j.PatternLayout

log4j.appender.archivo.layout.ConversionPattern = %d [%-5p] %c{2} - %m%n

log4j.rootLogger=TRACE, consola

log4j.logger.com.javatutoriales.log4j.configuracion=WARN, archivo

De la misma forma, es posible definir distintos appenders para distintos loggers. Suponiendo que la aplicación tiene dos paquetes, un paquete "com.javatutoriales.log4j.configuracion.datos" y "com.javatutoriales.log4j.configuracion.conexiones", se pueden asociar distintos loggers, así como colocarles distintos niveles de log, a estos paquetes de la siguiente forma:

log4j.logger.com.javatutoriales.log4j.configuracion.datos = WARN, archivo

log4j.logger.com.javatutoriales.log4j.configuracion.conexiones = INFO, consola

**Conclusiones**

Hemos llegado a la conclusión que las definiciones de log no solo ayudan a supervisar sino a mejorar el procesamiento de nuestros datos.

**Recomendaciones**

Estamos viendo que podemos mejorar más el tema de los log desarrollando seguimientos a cada aplicación o programa realizado en java POO en netbeans para poder mejorar nuestro desempeño laboral.

# Bibliografía

Jared, D. (2011). *Usar logs en Java parte 1.* Madrid: https://danjared.wordpress.com/2009/05/11/usar-logs-en-java-parte-1/.

Java, T. d. (2011). *Log4j para Creación de Eventos de Log.* Madrid: http://www.javatutoriales.com//2011/04/log4j-para-creacion-de-eventos-de-log.html.

**Anexos**

<https://danjared.wordpress.com/2009/05/11/usar-logs-en-java-parte-1/>.

https://danjared.wordpress.com/2009/05/11/usar-logs-en-java-parte-2/.

http://www.javatutoriales.com//2011/04/log4j-para-creacion-de-eventos-de-log.html.

https://www.youtube.com/watch?v=w1apI8VjaBU