

BITACORA DE MENSAJES CON LOG4J

1. GENERALIDADES- EVENTOS DE LOG

Dentro de la creación de software un componente que es útil es la información de seguimiento (líneas de rastreo) dentro del desarrollo, llamadas bitácoras o LOG, con el fin de verificar o garantizar el correcto funcionamiento de la aplicación. Esta información es diferente a los mensajes al usuario final.

Su implementación se puede realizar de varias maneras:

- MENSAJES DIRECTOS A CONSOLA,
- USANDO UN API.

1.1 MENSAJES DIRECTOS A CONSOLA

Estos son mensajes que se envían con la sentencia:

System.out.println();

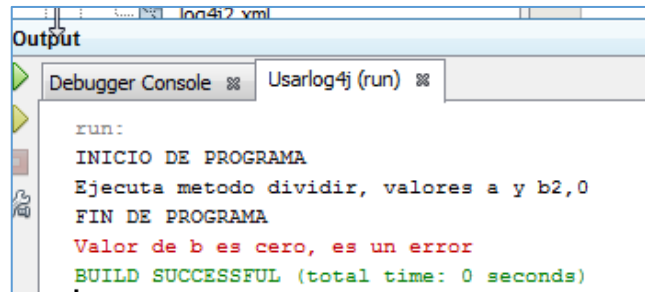
La ventaja de esta solución es que es rápida, pero el inconveniente es que tienen un uso específico y después si se olvida retirarlas, generan salidas a la consola que ya en tiempo de ejecución generan recarga en funcionamiento.

Acá un ejemplo:

```
/**
 * I
 * @author Vinni
 */
public class Operaciones {
    public static void main(String[] args) {
        Operaciones op =new Operaciones();
        System.out.println("INICIO DE PROGRAMA");
        int a = op.dividir(2, 1);
        System.out.println("FIN DE PROGRAMA");
    }
    public int dividir(int a, int b){
        System.out.println("Ejecuta metodo dividir, valores a y b"+a+", "+b);
        if (b==0){
            System.err.println("Valor de b es cero, es un error ");
        }
        return a/b;
    }
}
```

Se ve una clase que invoca un método dividir. Se han colocado 4 mensajes, 2 de tipo informativo, 1 de error y 1 de tipo consulta de valores.

La salida en la consola:



Acá se observa los mensajes de respuesta, y todo en mezclado. Los mensajes err. Aparecen en rojo y los demás en color negro.

1.2 USO DE API

Otra estrategia para generar una bitácora, se puede usar API's (Framework), que se especializan y brindan la facilidad de realizar seguimiento y filtrar los mensajes.

Los framework conocidos son: el Componente que viene incorporado dentro del JDK de Oracle® y el LOG4j de Apache®; estos ofrecen una forma jerárquica de colocar las sentencias de log dentro de una aplicación Java. Se tienen múltiples formatos de salida, y múltiples niveles de información de log.

Log4j tiene tres componentes principales:

Loggers

Appenders

Layouts

Estos tres tipos de componentes trabajan juntos para permitir a los desarrolladores hacer el log de mensajes de acuerdo al tipo y nivel de mensaje, controlar en tiempo de ejecución como es que estos mensajes son formateados y dónde son reportados.

1.2.1 LOGGERS

Los loggers son los responsables de capturar la información de logging. Los loggers son entidades con nombre. Los nombres de los loggers son case-sensitive y siguen una regla jerárquica de nombres. Ejemplo: el **logger "app.util.Operaciones"** es padre del **logger "app.util.Operaciones"**.

Normalmente se crea un logger por cada una de las clases de las cuales que se desea seguir, y logger tiene el mismo nombre que la clase.

Los loggers pueden tener niveles asignados. Los niveles normales que puede tener un logger son, de menor a mayor prioridad:

TRACE: usado para información más detallada que el nivel debug.

DEBUG: usado para mensajes de información detallada que son útiles para debugear una aplicación.

INFO: usado para mensajes de información que resaltan el progreso de la aplicación de una forma general.

WARN: usado a para situaciones que podrían ser potencialmente dañinas.

ERROR: usado eventos de error que podrían permitir que la aplicación continúe ejecutándose.

FATAL: usado para errores muy graves, que podrían hacer que la aplicación dejara de funcionar.

Las peticiones de logging se realizan usando los métodos del framework:

"trace", "debug", "info", "warn", "error", y "fatal".

Una petición de logging está habilitada si su nivel es mayor que o igual que el nivel de su logger, en el siguiente cuadro se ve con mayor claridad:

Nivel de los mensajes que se mostrarán					
Nivel del Logger	DEBUG	INFO	WARN	ERROR	FATAL
	DEBUG				
	INFO				
	WARN				
	ERROR				
	FATAL				
	ALL				
	OFF				

Un ejemplo es que si se establece el nivel de log en "DEBUG" se mostrarán los mensajes de nivel "DEBUG", "INFO", "WARN", "ERROR", y "FATAL". Si, se cambia a nivel en "ERROR", solo se mostrarán los mensajes de nivel "ERROR", y "FATAL".

Se pueden tener en cuenta dos niveles especiales:

ALL: Es el nivel más bajo posible y se usa para activar todo el logging.

OFF: Es el nivel más alto posible y se usa para evitar cualquier mensaje de log

1.2.2 APPENDERS

El framework log4j permite enviar los mensajes a múltiples destinos, y eso se llama appender. Existen appenders para la consola, archivos, sockets, JMS, correo electrónico, bases de datos, etc. Además también es posible crear appenders propios:

Los appenders disponibles son:

AsyncAppender: Permite enviar los mensajes de log de forma asíncrona. Este appender recolectará los mensajes enviados a los appenders que estén asociados a él y los enviará de forma asíncrona.

ConsoleAppender: Envía los eventos de log a System.out or System.err

FileAppender: Envía los eventos de log a un archivo

DailyRollingFileAppender: Extiende FileAppender de modo que el archive es creado nuevamente con una frecuencia elegida por el usuario

RollingFileAppender: Extiende FileAppender para respaldar los archivos de log cuando alcanza un tamaño determinado

ExternallyRolledFileAppender: Escucha en por un socket y cuando recibe un mensaje, lo agrega en un archivo, después envía una confirmación al emisor del mensaje.

JDBCAppender: Proporciona un mecanismo para enviar los mensajes de log a una base de datos.

JMSAppender: Publica mensajes de log como un tópico JMS.

LF5Appender: Envía los mensajes de log a una consola basada en swing.

NTEventLogAppender: Agrega los mensajes a un sistema de eventos NT (solo para Windows)

NullAppender: Un appender que no envía los mensajes a ningún lugar.

WriterAppender: Envía los eventos de log a un Writer o a un OutputStream dependiendo de la elección del usuario.

SMTPAppender: Envía un e-mail cuando ocurre un evento específico de logging, típicamente errores o errores fatales.

SocketAppender: Envía un objeto LoggingEvent a un servidor remoto, usualmente un SocketNode.

SocketHubAppender: Envía un objeto LoggingEvent a un servidor remoto de log.

SyslogAppender: Envía mensajes a un demonio syslog remoto




TelnetAppender: Es un appender que se especializa en escribir a un socket de solo lectura.

1.2.3 LAYOUTS

Estos componentes ayudan a dar formatos a los mensajes de salida, de acuerdo a lo que el usuario quiera; presentado el mensaje con formato para presentarlo en la consola o guardarlo en un archivo de texto, en una tabla HTML, un archivo XML, entre otros.

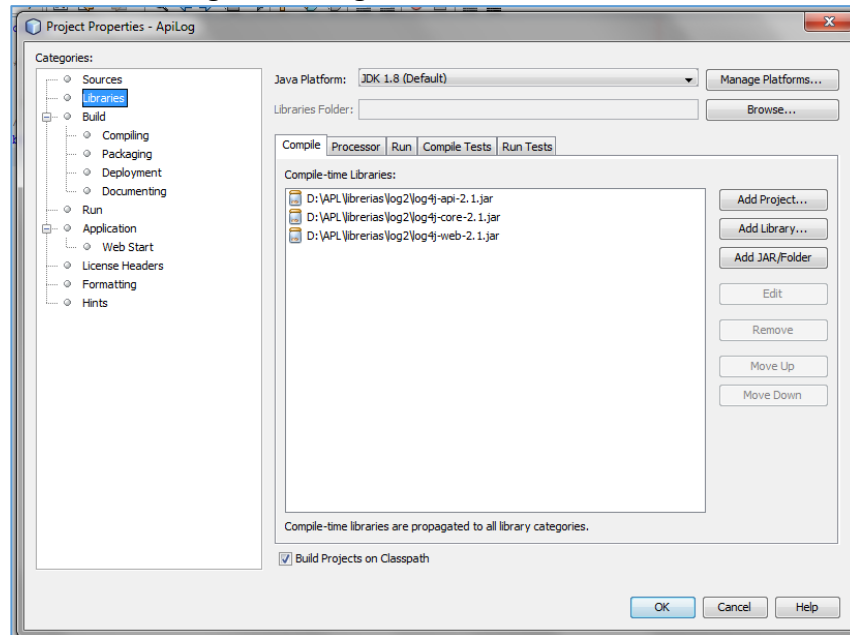
2. EJEMPLO DE LOG

En la implementación de una aplicación con el uso de LOG4j los pasos son:

1. Descargar la librería LOG4j versión 2.1, los archivos son ( log4j-core-2.1.jar  log4j-web-2.1.jar  log4j-api-2.1.jar).

Ubique los archivos en una ruta que se de fácil acceso para después referenciarlo en el proyecto.

2. Asociarlos a la aplicación. Dependiendo del IDE, adicione los archivos como librerías. En Netbeans en las propiedades del proyecto está la opción de librerías, como se muestra en la siguiente imagen.



3. Paso seguido, en la clase que se desea revisar, se debe crear el Logger, para eso se pueden usar una de las siguientes opciones:

- a. **`Logger log = Logger.getRootLogger();`**
- b. **`Logger log = Logger.getLogger("MiLogger");`**
- c. **`private static Logger log = Logger.getLogger(MiClase.class.getName());`**

Estas requieren de importar los objetos de:

`import org.apache.log4j.Logger;`

```
/**
 *
 * @author Vinni
 */
public class Operaciones {
    public static Logger logger = Logger.getLogger(Operaciones.class.getName());

    public static void main(String[] args) {
        Operaciones op = new Operaciones();
        logger.info("INICIO DE PROGRAMA");
        int a = op.dividir(2, 0);
        logger.info("FIN DE PROGRAMA");
    }

    public int dividir(int a, int b) {
        System.out.println("Ejecuta metodo dividir, valores a y b"+a+","+b);
        if (b==0) {
            logger.debug("Valor de b es cero, es un error ");
            b = 1;
        }
        return a/b;
    }
}
```

Para una implementación avanzada puede usar esta instancia:

a. ***private static final Logger LOG = LogManager.getLogger(MiClase.class);***

Se importan las clases de:

import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

```
package util;

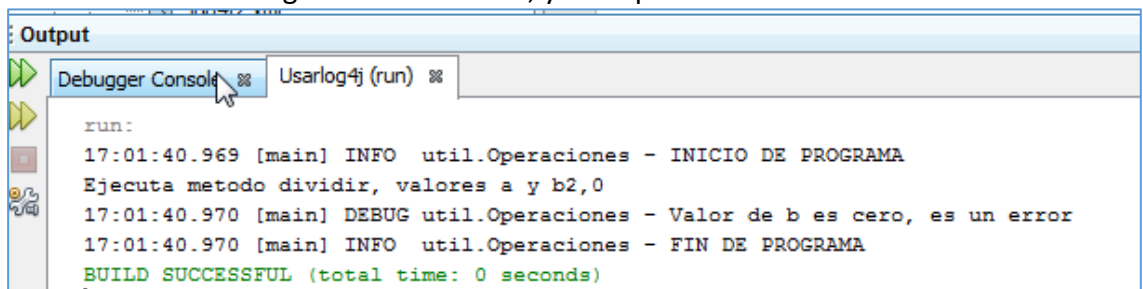
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

/**
 *
 * @author Vinni
 */
public class Operaciones {
    private static final Logger logger = LogManager.getLogger(Operaciones.class);

    public static void main(String[] args) {
        Operaciones op = new Operaciones();
        logger.info("INICIO DE PROGRAMA");
        int a = op.dividir(2, 0);
        logger.info("FIN DE PROGRAMA");
    }

    public int dividir(int a, int b){
        logger.debug("Ejecuta metodo dividir, valores {}, {}", a, b);
        if (b==0){
            logger.error("Valor de b es cero, es un error ");
            b= 1;
        }
        return a/b;
    }
}
```

4. Para la salida se configura el archivo xml, y la respuesta es:



```
run:
17:01:40.969 [main] INFO util.Operaciones - INICIO DE PROGRAMA
Ejecuta metodo dividir, valores a y b2,0
17:01:40.970 [main] DEBUG util.Operaciones - Valor de b es cero, es un error
17:01:40.970 [main] INFO util.Operaciones - FIN DE PROGRAMA
BUILD SUCCESSFUL (total time: 0 seconds)
```

ARCHIVO XML El nombre del archivo deberá ser: log4j2.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="OFF">
    <appenders>
```

```

<RollingFile name="Archivo" fileName="c:/demolog4j/logs/app.log"
    filePattern="logs/${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log">
    <PatternLayout>
        <pattern>%d %p %C{1.} [%t] %m%n</pattern>
    </PatternLayout>
    <Policies>
        <TimeBasedTriggeringPolicy />
        <SizeBasedTriggeringPolicy size="250 MB"/>
    </Policies>
    <DefaultRolloverStrategy max="20"/>
</RollingFile>
<Console name="Consola" target="SYSTEM_OUT">
    <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
    %msg%n"/>
</Console>
</appenders>
<loggers>
    <logger name="ArchivoLOG" level="ERROR" additivity="false">
        <appender-ref ref="Archivo"/>
    </logger>
    <root level="DEBUG">
        <appender-ref ref=" Consola"/>
    </root>
</loggers>
</configuration>

```

3. EJERCICIO

1. Replique el ejercicio y verifique su funcionalidad.
2. Construya una aplicación que reciba una cadena y que la guarde en un archivo. Y que reciba una cadena, nombre de archivo y cargue la información del archivo y visualiza en consola.
3. Aplique en la aplicación mensajes de Info, Debug y Error.