Ministerio de Educación
Universidad Tecnológica
Nacional Facultad Regional
Mendoza

TECNICATURA UNIVERSITARIA EN PROGRAMACION
Laboratorio II - Ciclo lectivo 2024

Características del Práctico : Log4J	
Unidad	**4**
Tema	**Edición y Depuración de Programas.**
Resultados de Aprendizajes	RA1: Aplica la herramienta Log4J para obtener la bitácora de ejecución de una aplicación Java.
Objetivo	El objetivo de este trabajo práctico es que los estudiantes adquieran un entendimiento profundo de la principal herramienta propuesta para logging.
Requisitos técnicos	Utilizar Spring Boot y Gradle como sistema de gestión de dependencias. Codificar una solución básica en Java.
Fecha de inicio	Junio de 2024 -
Fecha de entrega:	Junio de 2024
Modalidad de entrega :	Adjuntar el trabajo en formato Word al link correspondiente en el aula virtual.
Comisiones	**Comisiones: Mañana y tarde**
Modalidad del Trabajo Práctico	Desarrollo Grupal – Entrega en aula virtual
Conclusiones: Los estudiantes deben resumir lo que aprendieron durante la realización del trabajo práctico y destacar cómo las herramientas de logging permiten el seguimiento de la ejecución de una aplicación.	

<table>
<tr><td colspan="2"></td></tr>
<tr><td>Forma de Presentación</td><td>El trabajo debe presentarse en el link correspondiente al trabajo práctico alojado en el aula virtual.</td></tr>
<tr><td>Tipo de Evaluación</td><td>Formativa - Grupal</td></tr>
<tr><td>Modalidad de Evaluación</td><td>
<table>
<tr><td></td><td>100</td><td>>70%</td><td><50 %</td></tr>
<tr><td>Configuración del entorno</td><td></td><td></td><td></td></tr>
<tr><td>Configuración del archivo de propiedades</td><td></td><td></td><td></td></tr>
<tr><td>Codificación Java</td><td></td><td></td><td></td></tr>
<tr><td>RESULTADOS</td><td>Excelente</td><td>Satisfactorio</td><td>No Satisfactorio</td></tr>
</table>
Se establece una sumatoria de cada ítem y se determina el porcentaje individual de cada uno. Posteriormente se establece la sumatoria de todos los ítems y se lo divide por la cantidad de los mismos para determinar en cuál de los valores de la escala se encuadra.
</td></tr>
<tr><td>Bibliografía</td><td>Listas de reproducción de videos de la cátedra y material teórico del aula virtual</td></tr>
<tr><td>Profesores responsables del TP.</td><td>
<table>
<tr><td>Apellido y nombre</td><td>Apellido y nombre</td></tr>
</table>
</td></tr>
</table>

Ministerio de Educación
Universidad Tecnológica
Nacional Facultad Regional
Mendoza

	Julio Monetti	Claudia Naveda

DESARROLLO

PARTE A

Con el proyecto facilitado

Ejercicio 1

Investigue y explique con sus palabras qué produce la etiqueta **Loggers** en el archivo de configuración de log4j.

La etiqueta Loggers en el archivo de configuración de log4j se utiliza para definir las configuraciones de registro específicas para diferentes paquetes, clases o componentes de una aplicación.

Dentro de la etiqueta Loggers, se pueden definir múltiples elementos Logger, cada uno asociado con un nombre que representa el paquete, clase o componente al que se refiere. Estos elementos Logger permiten controlar el nivel de registro (por ejemplo, DEBUG, INFO, WARN, ERROR) y la configuración de los destinos de registro (como archivos, consola, bases de datos, etc.) para cada componente específico de la aplicación

TRACE: El nivel de registro más detallado. Se utiliza para mensajes extremadamente detallados y de seguimiento.
DEBUG: Se utiliza para mensajes de depuración que son útiles durante el desarrollo y pruebas del software.
INFO: Se utiliza para mensajes informativos que no son de depuración pero que proporcionan información relevante sobre el funcionamiento de la aplicación.
WARN: Se utiliza para mensajes que indican posibles problemas o situaciones inesperadas que no son necesariamente errores.
ERROR: Se utiliza para mensajes que indican errores que han ocurrido en la aplicación. Estos mensajes representan condiciones de error que deben ser investigadas y corregidas.
FATAL: El nivel más crítico. Se utiliza para mensajes que indican errores graves que probablemente causarán que la aplicación termine de manera inmediata.

Ejercicio 2

Investigue y explique con sus palabras qué produce la etiqueta **Appenders** en el archivo de configuración de log4j.

En Log4j, los appenders se utilizan para reenviar mensajes de registro a diferentes destinos. Log4j viene con múltiples complementos listos para usar, lo que permite enviar mensajes de registro a la consola, archivos, bases de datos, etc.

Reenvío de mensajes con Appenders:

name: Se utiliza para identificar de manera única el appender dentro de la configuración del registro.

target: Especifica hacia dónde se dirigirán los registros generados por el appender.

filter: Determina si este appender debe aceptar un mensaje de registro.

layout: Define el formato y patrón que debe seguir el mensaje de registro.

follow: Determina si el appender respeta las reasignaciones de System.out o System.err usando System.setOut o System.setErr.

direct: Si se establece en verdadero, este appender escribirá directamente en java.io.FileDescriptor y omitirá System.out. Puede aumentar el rendimiento hasta 10 veces cuando la salida se redirige a un archivo u otro proceso.

ignoreExceptions: Determina si este appender ignorará o no las excepciones. El valor predeterminado es verdadero y, si se establece en falso, las excepciones se propagarán.

Ejercicio 3

Investigue y explique con sus palabras que produce la etiqueta **Root Level** en el archivo de configuración de log4j.

El root level determina que logs se van a mostrar en consola o se guardaran en un archivo en especifico.

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

   <appender name="console" class="org.apache.log4j.ConsoleAppender">
      <layout class="org.apache.log4j.PatternLayout">
         <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n" />
      </layout>
   </appender>

   <root>
      <priority value="INFO" />
      <appender-ref ref="console" />
   </root>

</log4j:configuration>
```

En este caso el <root> <priority> nos dice que se mostraran y guardaran los logs de tipo INFO, este mensaje se puede cambiar por ejemplo por WARNING, ERROR, e.t.c.

Y en <appender-ref reg = ””> nos dice en donde se mostrar y/o guardaran los logs.

En el siguiente caso, se ve como se guardará en una dirección de archivo, y también se mostrará en consola todos los tipos de logs que pueden ocurrir.

```
<root>
   <priority value="ALL" />
   <appender-ref ref="consoleAppender" />
   <appender-ref ref="fileAppender" />
</root>
```

Ejercicio 4

Investigue y documente: qué es Apache Flume ?

Apache Flume es un servicio de distribución de datos y registro que permite la recopilación, agregación y movimiento de grandes cantidades de datos de forma eficiente. Se utiliza comúnmente en entornos de big data para recolectar datos de múltiples fuentes y enviarlos a un sistema de almacenamiento centralizado como Hadoop.

1. Casos de uso comunes:
    - Recopilación de logs de servidores y aplicaciones.
    - Ingesta de datos en tiempo real desde sensores y dispositivos IoT.
    - Integración de datos de múltiples fuentes para análisis de big data.
    - Monitoreo de la actividad en redes y sistemas.

2. Características principales:
    - Escalabilidad: Apache Flume puede manejar grandes volúmenes de datos y escalar horizontalmente según sea necesario.
    - Confiabilidad: Permite la replicación de datos para garantizar la integridad y disponibilidad de los mismos.
    - Flexibilidad: Puede conectarse con una variedad de fuentes de datos y destinos, como logs, bases de datos, sistemas de mensajería, entre otros.
    - Tolerancia a fallos: Flume está diseñado para manejar fallos de forma robusta, permitiendo la recuperación y reenvío de datos en caso de interrupciones.

PARTE B

Ejercicio

1. Creé una aplicación Java llamada **milog**. La misma debe contar solamente con la clase principal, la cual expone un mensaje sobre la consola informando que la aplicación se ha iniciado.

```java
package org.example.Punto1;
import java.util.logging.Logger;
public class Milog {
   private static final Logger logger =
Logger.getLogger(Milog.class.getName());
   public static void main(String[] args) {
       logger.info("La aplicación se ha iniciado.");
   }
}
```

Ejercicio

2. Cree una aplicación que utilice un nivel de logging personalizado, denominado **miLogLevel**.

   Indique cómo crear el nuevo nivel de logging desde el archivo de configuración.

   1-Abre o crea el archivo de configuración: Debes abrir o crear un archivo de configuración de Log4j2. Este archivo puede ser log4j2.xml.
   2-Define el nuevo nivel de logging: Dentro del bloque de configuración <Loggers>, define tu nuevo nivel de logging utilizando <Level>. Asigna un nombre único para tu nuevo nivel y especifica un valor numérico para él.
   3-Guarda los cambios: Guarda el archivo de configuración una vez que hayas definido el nuevo nivel de logging.
   4-Utiliza el nuevo nivel de logging en tu código: Ahora puedes utilizar este nuevo nivel de logging en tu código de la misma manera que utilizas los niveles de logging estándar.

```
package org.example.Punto2;
import java.util.logging.*;
public class LogginPersonalizado extends Level{
   public static final Level MI_LOG_LEVEL = new
LogginPersonalizado("MI_LOG_LEVEL",
Level.INFO.intValue() + 1);
   protected LogginPersonalizado(String name, int
value) {
       super(name, value);
   }
}
package org.example.Punto2;
import java.util.logging.Logger;
public class Main {
   private static final Logger logger =
Logger.getLogger(Main.class.getName());
   public static void main(String[] args) {
       logger.log(LogginPersonalizado.MI_LOG_LEVEL,
"Mensaje de mi log personalizado");
   }
}
```

Ejercicio

3. Cree una aplicación que envíe mensajes de logging a:
    a. Consola
    b. Un fichero denominado **log.txt**.
    c. Investigue y aplique el logging sobre http.

```
package org.example.Punto3;
import java.io.IOException;
import java.util.logging.*;
public class Main {
          private   static   final   Logger   logger   =
Logger.getLogger(Main.class.getName());
   public static void main(String[] args) {

       /* punto a
       logger.info("Mensaje a Consola");
       */


       //punto b
        try {
                          FileHandler  fileHandler  =  new
FileHandler("log.txt");
           fileHandler.setFormatter(new SimpleFormatter());
           logger.addHandler(fileHandler);
       } catch (IOException e) {
```

```
            logger.log(Level.SEVERE, "Error al configurar el
manejador de archivo", e);
        }

        logger.info("Este es un mensaje de INFO.");
        logger.warning("Este es un mensaje de WARNING.");
    }
}
```

Loggin sobre http

```
<!-- Definir un appender para registrar solicitudes HTTP -->
    <appender name="HTTP" class="ch.qos.logback.core.FileAppender">
        <file>http.log</file>
        <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout class="ch.qos.logback.contrib.json.classic.JsonLayout"/>
        </encoder>
    </appender>

    <!-- Configurar el nivel de log para el appender HTTP -->
    <logger name="org.springframework.web" level="INFO">
        <appender-ref ref="HTTP"/>
    </logger>

    <!-- Configurar el nivel de log general -->
    <root level="INFO">
        <appender-ref ref="STDOUT"/>
    </root>

</configuration>
```

Los logging HTTP son un tipo especializado de registro utilizado para registrar solicitudes y respuestas HTTP en aplicaciones web. Estos registros son útiles para el monitoreo, la depuración y el análisis del rendimiento de aplicaciones web. En Java, los frameworks de logging como Log4j y Logback pueden ser configurados para registrar solicitudes y respuestas HTTP.

Aquí hay un ejemplo de cómo configurar Logback, un framework de logging muy popular, para registrar solicitudes y respuestas HTTP en una aplicación web utilizando el framework Spring Boot

Ejercicio

4. ¿Qué produce el siguiente appender ? Codifique y de un ejemplo de salida

```
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <HTMLLayout>
      </HTMLLayout>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

```java
package org.example.EjercicioCuatro;

import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.core.config.Configurator;

public class Main {

   private static final Logger logger =
LogManager.getLogger(Main.class);

   public static void main(String[] args) {
       String xmlFileName = "log4j2Ej4.xml";
       Configurator.initialize(null, xmlFileName);

       logger.info("Este es un mensaje de informacion.");
   }
}
```

Ejercicio

5. Cree una aplicación que envíe mensajes de logging a un fichero, el cual es rotado cada 2 minutos.

XML:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Configuration status="INFO">
   <Appenders>
       <RollingFile name="RollingFile"
fileName="logs/mylogfile.log"

filePattern="logs/mylogfile-%d{yyyy-MM-dd-HH-mm}.log">
           <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss}
[%t] %-5level %logger{36} - %msg%n"/>
           <Policies>
               <TimeBasedTriggeringPolicy interval="2"
modulate="true"/>
           </Policies>
       </RollingFile>
   </Appenders>
   <Loggers>
       <Root level="info">
           <AppenderRef ref="RollingFile"/>
       </Root>
   </Loggers>
</Configuration>
```

Main:

```
package org.example.EjercicioCinco;

import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.core.config.Configurator;

public class Main {

   private static final Logger logger =
LogManager.getLogger(org.example.EjercicioCuatro.Main.class
);

   public static void main(String[] args) {
       String xmlFileName = "log4j2Ej5.xml";
       Configurator.initialize(null, xmlFileName);

       for (int i = 0; i < 10; i++) {
           logger.info("Mensaje de registro de ejemplo
número " + (i + 1));
           try {
               Thread.sleep(10000); // Espera 10 segundos
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
       }
```

```
    }
}
```

Ejercicio

6. Modifique la aplicación anterior para que el fichero sea rotado según una expresión cron. (Coloque un horario válido para probar que funciona)

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
   <Appenders>
       <RollingFile name="file"
fileName="C:\Users\andyb\OneDrive\Escritorio\logsPunto6.txt
"

filePattern="C:\Users\andyb\OneDrive\Escritorio\logsPunto6-
%d{yyyy-MM-dd_HH-mm}.txt">
           <PatternLayout pattern="%d{yyyy-MM-dd
HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
           <Policies>


               <TimeBasedTriggeringPolicy />
           </Policies>
       </RollingFile>
   </Appenders>
   <Loggers>
       <Root level="debug">
           <AppenderRef ref="file" />
       </Root>
   </Loggers>
</Configuration>
```

Ejercicio

7. Cree una aplicación para mostrar mensajes de logging al menos en cuatro formatos diferentes utilizando **PatternLayout**.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
   <Appenders>
       <!-- Consola :  Muestra solo el nivel y el mensaje
-->
       <Console name="Console" target="SYSTEM_OUT">
           <PatternLayout pattern="%-5level - %msg%n"/>
       </Console>

       <!-- Consola 2:  Muestra la fecha, el nivel y el
mensaje -->
       <Console name="Console2" target="SYSTEM_OUT">
           <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t]
%-5level - %msg%n"/>
       </Console>

       <!-- Consola 3:  Muestra el nivel, el nombre del
logger y el mensaje -->
       <Console name="Console3" target="SYSTEM_OUT">
           <PatternLayout pattern="%-5level [%logger] -
%msg%n"/>
       </Console>

       <!-- Consola 4: Muestra la fecha, el nivel, el
nombre del logger y el mensaje -->
       <Console name="Console4" target="SYSTEM_OUT">
           <PatternLayout pattern="%d{yyyy-MM-dd
HH:mm:ss.SSS} [%t] %-5level [%logger] - %msg%n"/>
       </Console>
   </Appenders>
   <Loggers>

       <!-- PARA VER LOS DEMÁS PATTERN LAYOUT SE DEBE
REMPLAZAR EL NOMBRE DE LA CONSOLA EN ref = "Console", por
alguno de
        las consolas creadas en los appenders -->

       <Root level="trace">
           <AppenderRef ref="Console"/>
       </Root>




   </Loggers>
</Configuration>
```

Ejercicio

8. Qué resultado produce el siguiente patrón:
    a. %d [%-6p] %c{1} - %m%n

Si modifico el patternLayaout me da como resultado la fecha y la hora, además de su mensaje:

2024-04-25 08:29:00,972 [INFO] Main - Este es un mensaje de información
2024-04-25 08:29:00,979 [DEBUG] Main - Este es un mensaje de depuración
2024-04-25 08:29:00,980 [TRACE] Main - Este es un mensaje de traza

    b. %sn %d{yyyy/MM/dd HH:mm:ss,SSS} %r [%-6p] [%t] %c{3} %C{3}.%M(%F:%L) - %m%n

Muestra la línea donde fue impreso el logger junto con su mensaje.

1 2024/04/25 08:30:55,039 928 [INFO] [main] org.example.Main EjercicioOcho.Main.main(Main.java:14) - LA APLICACIÓN SE HA INICIADO

2 2024/04/25 08:30:55,043 932 [ERROR] [main] org.example.Main EjercicioOcho.Main.main(Main.java:15) - ERROR EN LA LINEA

Ejercicio

9. Investigue, documente y aplique un ejemplo de la clase ThreadContext.

Un ThreadContext es un mapa de encabezados de cadena y un mapa transitorio de objetos con clave asociados con un hilo. Permite almacenar y recuperar información de encabezado a través de llamadas a métodos, llamadas de red y subprocesos generados a partir de un subproceso que tiene un ThreadContextasociado. Los subprocesos generados a partir de un ThreadPoolsoporte listo para usar ThreadContexty todos los subprocesos generados heredarán el ThreadContextsubproceso del que se bifurca". Las llamadas de red también conservarán los encabezados de los remitentes automáticamente.

Los consumidores de ThreadContext generalmente no necesitan interactuar agregando o guardando contextos. Cada subproceso de elasticsearch es administrado por un grupo de subprocesos o un ejecutor que es responsable de almacenar y restaurar el contexto de los subprocesos.

Por ejemplo, si se recibe una solicitud de red, todos los encabezados se deserializan de la red y se agregan directamente como encabezados de los subprocesos ThreadContext(consulte readHeaders(StreamInput). Para no modificar el contexto que está actualmente activo en este subproceso, el código de red utiliza un try/with patrón para ocultar su contexto actual, leer los encabezados en uno nuevo y una vez que se maneja la solicitud o se bifurca un hilo de controlador (que a su vez hereda el contexto), restaura el contexto anterior.

En el siguiente ejemplo podemos observar como crearemos 3 hilos con su propio ID con un `ThreadContext.put("sessionId", sessionId)`,esto hace que cada hilo posea un ID en la memoría, el cual será pedido más adelante, A esto nos referimos cuando hablamos del contexto de los Threads, nos sirve para llevar registro especifico de cada uno de los hilos que vayamos a ejecutar.

Luego, al final del ejemplo se utiliza un `ThreadContext.clearMap();,` para que todo dato que hayamos guardado en la RAM acerca de los Threads, no se quede registrada en la memoria y así evitar un desbordamiento en esta.

```
  public static void main(String[] args) {

        for (int i = 1; i <= 3; i++) {
            int taskId = i;
            Thread thread = new Thread(() -> {

                String sessionId = "Session-" + taskId;

                ThreadContext.put("sessionId", sessionId);

                logger.info("Inicio de tarea " + taskId);
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                logger.info("Fin de tarea " + taskId);

                ThreadContext.clearMap();
            });
            thread.start();
        }
    }
}
```