Logging Using Log4j 2

Document Revision

Author	**Mohamed Sabri**
Date	**9th May 2018**
Version	**1.0**

Contents

What is Log4j 2

log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License. log4j is a popular logging package written in Java. log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages.

Apache Log4j 2 is an upgrade to Log4j 1.x that provides significant improvements such as performance improvement, automatic reloading of modified configuration files, java 8 lambda support and custom log levels.

Basic Log4j2 Configuration

To start using log4j2 in your project, you simply need to add the log4j-core and log4j-api maven dependencies to the pom.xml file of the project (HMI-Parent).

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.0.2</version>
</dependency>

<dependency>
     <groupId>org.apache.logging.log4j</groupId>
     <artifactId>log4j-core</artifactId>
     <version>2.0.2</version>
</dependency>
```

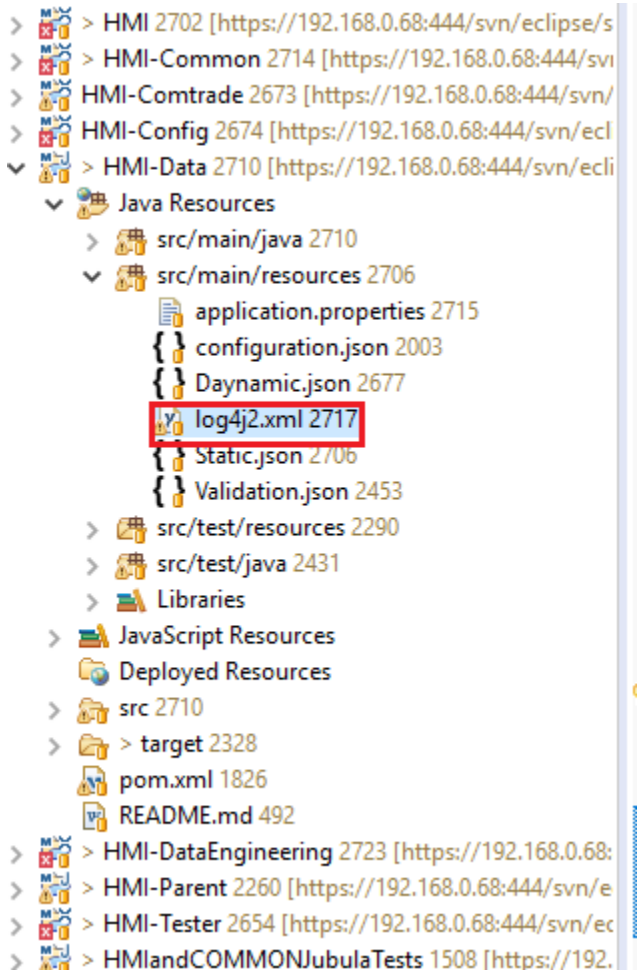
log4j 2 will automatically provide a simple configuration, if you don't explicitly define one yourself. The default configuration logs to the console at a level of ERROR level or above.

Customizing the Log4j 2 Configuration

A custom log4j2 configuration can be created either programmatically or through a configuration file.

The library supports configuration files written in XML, JSON, YAML, as well as the *.properties* format.

First, you can override the default configuration by simply creating a *log4j2.xml* file on the classpath. Log4j will scan all classpath locations to find out this file and then load it.



Configuration File (log4j2.xml) Tags

1- Properties Tag

```
<Properties>
    <Property name="basePath">logs</Property>
</Properties>
```

Make a property of the path where the log files will be created in the file system , here the files will be created at the same path of the project on a folder named logs.

2- Appenders Tag

Appenders are responsible for delivering LogEvents to their destination.

```
<Appenders>

    <RollingFile name="hmiDebugFileAppender" fileName="${basePath}/hmi-debug.log"
        filePattern="${basePath}/hmi-debug-%d{yyyy-MM-dd}.log">
        <Filters>
            <ThresholdFilter level="info" onMatch="DENY"
                onMismatch="NEUTRAL" />
            <ThresholdFilter level="debug" onMatch="ACCEPT"
                onMismatch="DENY" />
        </Filters>
        <PatternLayout>
            <pattern>[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n
            </pattern>
        </PatternLayout>
        <Policies>
            <SizeBasedTriggeringPolicy size="50 MB"></SizeBasedTriggeringPolicy>
        </Policies>
        <DefaultRolloverStrategy max="20" />
    </RollingFile>

    <Console name="console" target="SYSTEM_OUT">
         <PatternLayout   pattern="[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n" />
    </Console>

</Appenders>
```

3- Loggers Tag

Loggers tag contains a list of Logger instances. The Root element is a standard logger that outputs all messages and it is mandatory in every configuration.

If you don’t provide a Root element , it will automatically be configured by default with a Console appender and the ERROR log level.

```
<Loggers>
    <Logger name="HMI" level="debug" additivity="true">
        <AppenderRef ref="hmiDebugFileAppender" level="debug" />
        <AppenderRef ref="hmiInfoFileAppender" level="info" />
        <AppenderRef ref="hmiErrorFileAppender" level="error" />
    </Logger>

    <Root level="debug" additivity="false">
        <!--  <appender-ref ref="console" />  -->
    </Root>
</Loggers>
```

Configuring Appenders

In the log4j2 architecture, an appender is basically responsible for sending log messages to a certain output destination (console,file,database,etc).

Here are some of the most useful types of appenders that the library provides:

- ConsoleAppender: logs messages to the System console.
- FileAppender : writes log messages to a file.
- RollingFileAppender: writes the messages to a rolling log file.
- JDBCAppender: uses a relational database for logs.

We used the RollingFileAppender in our code .

The RollingFileAppender:

```
<Appenders>
    <RollingFile name="hmiDebugFileAppender" fileName="${basePath}/hmi-debug.log"
        filePattern="${basePath}/hmi-debug-%d{yyyy-MM-dd}.log">
```

The RollingFileAppender is an OutputStreamAppender that writes to the File named in the fileName parameter and rolls the file over according the TriggeringPolicy and the RolloverPolicy and the new file will be named as the file pattern attribute .

The rolling file appender can use many types of ploicies like :

- ***OnStartupTriggeringPolicy***: a new log file is created every time the JVM starts.
- ***TimeBasedTriggeringPolicy***: the log file is rolled based on a date/time pattern.
- ***SizeBasedTriggeringPolicy:*** the file is rolled when it reaches a certain size.

```
<Policies>
    <SizeBasedTriggeringPolicy size="50 MB"></SizeBasedTriggeringPolicy>
</Policies>
<DefaultRolloverStrategy max="20" />
```

The SizeBasedTriggeringPolicy causes a rollover once the file has reached the specified size which is 50 MB in the screen shot example.

The DefaultRolloverStrategy max parameter defines the maximum value of the counter. Once this values is reached older archives will be deleted on subsequent rollovers. The default value is 7 and we change it to be 20.

Configuring Layouts

layouts are used by appenders to define how a log message will be formatted.

- *PatternLayout*: configures messages according to a String pattern.
- *JsonLayout*: defines a JSON format for log messages.
- *CsvLayout*: can be used to create messages in a CSV format.

The PatternLayout:

The mechanism is primarily driven by a conversion pattern which contains conversion specifiers. Each specifier begins with the % sign, followed by modifiers that control things like width and color of the message, and a conversion character that represents the content, such as date or thread name.

```
<PatternLayout>
    <pattern>[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n
    </pattern>
</PatternLayout>
```

- *%level*: displays the log level of the message (all,trace,debug,info,warn,error,..)
- *%d{HH:mm:ss.SSS}:* outputs the date of the log event in the specified format.
- *%t:* outputs the thread name.
- *%c{1}:* outputs the name of the logger that published the logging event.
- *%msg%n*: outputs the log message.

Configuring Filters

Filters in log4j2 are used to determine if a log message should be processed or skipped.

A filter can be configured for the entire configuration or at logger or appender level.

There are many filters to use like ThresholdFilter, *BurstFilter* , *DynamicThresholdFilter* , RegexFilter , etc .

We just used ThresholdFilter on the appender level .

```
<Filters>
    <ThresholdFilter level="info" onMatch="DENY"
        onMismatch="NEUTRAL" />
    <ThresholdFilter level="debug" onMatch="ACCEPT"
        onMismatch="DENY" />
</Filters>
```

Level: A valid Level name to match on.

onMatch: Action to take when the filter matches. May be ACCEPT, DENY or NEUTRAL. The default value is NEUTRAL.

onMismatch: Action to take when the filter does not match. May be ACCEPT, DENY or NEUTRAL. The default value is DENY.

The example in the picture is an appender that accept debug level only , we already know that the levels by order are (All < Trace < Debug < Info < Warn < Error < Fatal)

So we make the ThresholdFilter that accept the debug level which will accept the debug and all levels less than debug , so we must ignore these levels so we make another ThresholdFilter to deny the info level which will deny info and all levels less than info , then we will have the filter which accept Debug level only .

We should put the ThresholdFilter for the deny above the ThresholdFilter for the accept in the configuration file.

Configuring Loggers

Besides the *Root* logger, we can also define additional *Logger* elements with different log levels, appenders or filters. Each *Logger* requires a name that can be used later to reference it in java code.

Using Loggers in Code:

- *Logger Configuration in log4j2.xml*

```
<Loggers>
    <Logger name="HMI" level="debug" additivity="true">
        <AppenderRef ref="hmiDebugFileAppender" level="debug" />
        <AppenderRef ref="hmiInfoFileAppender" level="info" />
        <AppenderRef ref="hmiErrorFileAppender" level="error" />
    </Logger>

    <Root level="debug" additivity="false">
        <!--  <appender-ref ref="console" />  -->
    </Root>
</Loggers>
```

1. Create a logger by using the <Logger > element.

2. Give that logger a name attribute which will use in java and a level attribute.

3. Reference the appenders you need to the logger by the <AppenderRef /> element.

- *Logger Creation in java*

```java
public class LoggerUtil {

    /**
     * prevent any class to make instance from the LoggerUtil class.
     */
    private LoggerUtil() {
    }

    private static Logger LOGGER = LogManager.getLogger(LoggerUtil.class.getName());

    public static void setLogger(String loggerName) {
        LOGGER = LogManager.getLogger(loggerName);
    }

    public static void trace(String message) {
        LOGGER.trace(message);
    }

    public static void debug(String message) {
        LOGGER.debug(message);

    }
```

1. Create LoggerUtil class that has one static Logger and write static methods for all logging levels which use your static logger .

2. You can get a specified logger from your configuration log4j2.xml by passing its name to the method LogManager.getLogger(logger_name).

3. Make a setLogger() method which take the logger name as parameter and get that logger then assign it to your static object Logger so you can used it.

```java
public static final String HMI_MODULE = "HMI";

public static void main(String[] args) {
    LoggerUtil.setLogger(HMI_MODULE);
    LoggerUtil.info(" now Logging using HMI Logger");
```

4. Before you use the logger you should set the logger using setLogger() method , then use the class LoggerUtil to call any of its static method to begin logging using the logger that you have been set.

Note that the logger name is the same string for the attribute name in the logger element in log42.xml configuration file.

References

Configuring Log4j2 ThresholdFilters.

Log4j 2 Tutorial and Examples

How Log4J2 Works