http://www.egovframe.go.kr/wiki/doku.php?id=egovframework:rte3:fdl:logging:log4j_2:%EC%84%A4%EC%A0%95_%ED%8C%8C%EC%9D%BC%EC%9D%84_%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94_%EB%B0%A9%EB%B2%95

Log4j 2 Configurations [configuration by file]

Outline

In Log4j 2, you do not have the conventional property files. Instead,
the newer configuration file formats such as XML (log4j2.xml) and JSON (log4j2.json or log4j2.jsn) are available.

Refer to the following XML file for configuration of the properties using XML files and the Log4j 2 Manual for JSON.

Description

Log4j 2 XML Configuration

XML Directory

Create the XML file (log4j2.xml) in WEB-INF/classes.
Read the foregoing configuration file when Log4j 2 is initialized.

Definition of XML File

In Log4j 2, the element **<Configuration>** is at the uppermost hierarchy.
You can configure the sub-elements such as Logger, Appender and Layout under <Configuration>.

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

  <!-- Configuring Appender and Layout -->
  <Appenders>
    <Console name="console" target="SYSTEM_OUT">
      <PatternLayout/>
    </Console>
    <File name="file" fileName="./logs/file/sample.log" append="false">
      <PatternLayout pattern="%d %5p [%c] %m%n"/>
    </File>
  </Appenders>

  <!-- Configuring Logger -->
  <Loggers>
```

```
    <Logger name="egovLogger" level="DEBUG" additivity="false">
      <AppenderRef ref="console"/>
      <AppenderRef ref="file"/>
    </Logger>
    <Rootlevel="ERROR">
      <AppenderRef ref="console"/>
    </Root>
  </Loggers>

</Configuration>
```

References

[Log4j 2 Configuration](#)
[XML Syntax](#)
[JSON Syntax](#)

Configuring Logger

Being the subject of Log4j, Logger serves you a highly effective means with which you can proceed with the logging, save for logger configuration.
The first thing you need to do is to define the logger to be used within the application, followed by configuration of the log level and appender for definition of output directory and position.

Declaring and Defining Logger

According to the pre-defined hierarchy, you can declare the loggers, inclusive of Root Logger, in **<Loggers>**.
Root Logger and General Logger are defined as **<Root>** and **<Logger>**, respectively.
You can define one or more Loggers, provided you have defined Root Element.

```
  <Loggers>
    <Logger>...</Logger>
    <Root>...</Root>
  </Loggers>
  <Loggers>
    <!-- attribute: name(Log name), level(Log Level), additivity(Log-in additivity, true or false)
-->
    <!-- element: AppenderRef -->
    <Logger name="X.Y" level="INFO" additivity="false">
      <AppenderRef ref="console"/>
    </Logger>
    <Logger name="X" level="DEBUG" additivity="false">
      <AppenderRef ref="console"/>
    </Logger>
    <Rootlevel="ERROR">
      <AppenderRef ref="console"/>
    </Root>
  </Loggers>
```

Where the Appender “console” is not available in the foregoing AppenderRef, you are unable to proceed with the logging process.

Calling Logger

You can call the Logger in a code as follows:

```
    package egovframe.sample;

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

    public class LoggerTest {

      // (1) Generating the object Logger that follows configuration entitled
"egovframe.sample.LoggerTest"
      Logger logger1 = LogManager.getLogger();
      // (2) Ditto
      Logger logger2 = LogManager.getLogger(LoggerTest.class);
      // (3) Generating the Logger object that follows the configuration of Logger entitled
Logger Name "X"
      Logger logger3 = LogManager.getLogger("X");

    }
```

When the Logger is not available in the configuration file, as you can refer to in (1) and (2), refer to the following Logger Hierarchy for more information.
This signifies that Root Logger configuration must be followed to generate the Logger object generated in (1) and (2).

Logger Hierarchy

To better understand which logger object follows which configuration, you need to make doubly sure how Logger Hierarchy is made of.
Check out the object LoggerConfig is generated by the Logger configuration defined in the configuration file that establishes parent-child objects. The parent logger inherits the configurations unique to it to the child logger just like that.
For instance, the Logger “X.Y” has the parent Logger “X” whose parent logger is the Root Logger (Tier 1 Hierarchy).

Refer to the following description for how hierarchy is defined:

1) When the Logger having exactly the same name with the called Logger exists, the configuration of the existing Logger is inherited.
2) When the Logger not having exactly the same name with the called Logger but having a Parent Logger exists, the configuration of the Parent Logger is inherited.
3) When the Parent Logger does not exist, the Root Logger configuration is inherited.

Logger Name	Assigned LoggerConfig	Level	Java Code	Description
root	root	ERROR	LogManager.getLogger("root");	Root configuration counts
X	X	DEBUG	LogManager.getLogger("X");	X Logger counts
X.Y	X.Y	INFO	LogManager.getLogger("X.Y");	X.Y Logger counts
X.Y.Z	X.Y	INFO	LogManager.getLogger("X.Y.Z");	X.Y.Z Logger not available. X.Y (parent configuration) counts
X.YZ	X	DEBUG	LogManager.getLogger("X.YZ");	X.YZ Logger not available. X (parent configuration) counts
Y	root	ERROR	LogManager.getLogger("Y");	Y Logger configuration not available. Root configuration counts

Log Level

Log4j 2 provides the log levels FATAL, ERROR, WARN, INFO, DEBUG and TRACE.
You can output the logs using such logging methods as trace(), debug(), info(), warn(), error() and fatal().
Refer to the following for how log level is defined: (FATAL > ERROR > WARN > INFO > DEBUG > TRACE)

Log Level	Description
FATAL	Fatal error occurred. Fatal error occurred in the system that hinders execution of application. Barely used in the course of developing applications.
ERROR	Error occurred when the request is in process.
WARN	Warning that is not perceived as an error but potentially gives rise to system error in the near future.
INFO	Informative messages such as log-in and status change.
DEBUG	Messages used for debugging.
TRACE	Traces details in case of an extensive debug level.

You may change the log level while application is in operation. For reference to Logger Configuration, you need to case org.apache.logging.log4j.Logger to org.apache.logging.log4j.core.Logger to call the method that better refers to the logger configuration information.

In the event you need to change the Log Level, the Method setLevel() should be called in the concerned parameter.
Log Level is changed accordingly from the time whewn setLevel() is called. The Log Event below the designated level is to be ignored.

```
  package egovframe.sample;

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

  public class LoggerTest {
    Logger logger = LogManager.getLogger(); // Root Logger counts, Log Level: ERROR
    org.apache.logging.log4j.core.Logger targetLogger =
(org.apache.logging.log4j.core.Logger) logger;

    targetLogger.debug("Before - debug"); // Output
    targetLogger.error("Before    - error"); // No output


    targetLogger.setLevel(Level.DEBUG); // DEBUG, INFO, WARN, ERROR, FATAL
available
    targetLogger.debug("After - debug"); // Output
    targetLogger.error("After - error"); // Output
}
```

With no pre-processor available in Java, you may not split debugging codes for debugging and release. The code #ifdef DEBUG is not available in Java just like that, contrary to C Language. Using the codes wisely for administration of logs in Log4j is thus advised.

References

See Log4j 2 Logger for configuration details.

Configuring Appender

Appender states the directory of log output.
You can have the first look to presume how the class directory is defined, by referring to the name of a class such as XXXAppender.

Meanwhile, Log4j 2 supports various log output destinations and methods such as Console, File, RollingFile, Socket and DB.
Contrary to the conventional Log4j 1.x that uses class properties to refer to the Appender types, Log4j 2 uses tags.

Declaring and Defining Appender

This Section describes how to declare and define the most commonly used Appenders such as Console, File, RollingFile and JDBC Appender.
Type of Appender and tag for configuration vary by the output destination. See the following Table for a brief look:

Appender	Tag	Location
ConsoleAppneder	<Console>	Console

FileAppneder	<File>	File
RollingFileAppneder	<RollingFile>	File, conditional
JDBCAppender	<JDBC>	RDB Table

Appenders are to be declared under the high-tier **<Appenders>**.

```
  <Appenders>
    <Console>...</Console>
    <File>...</File>
    <RollingFile>...</RollingFile>
    <JDBC>...</JDBC>
  </Appenders>
```

Appender features the property ‘name’, where the title of Appender is to be designated. The property ‘name’ is referred to when Appender refers to the log output. Also defined in Appender is the output pattern layout.
Refer to the following for how to work the code out using ConsoleAppender and PatternLayout:

```
  <Appenders>
      <Console name="console" target="SYSTEM_OUT">
        <PatternLayout /> <!-- Default Pattern applied, %d{HH:mm:ss.SSS} [%thread] %-
5level %logger{36} - %msg%n -->
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="egovLogger" level="DEBUG" additivity="false">
      <AppenderRef ref="console" />
    </Logger>
    <Rootlevel="ERROR">
      <AppenderRef ref="console" />
    </Root>
  </Loggers>
```

Appender Types

Configurations to define Appender are as follows:

ConsoleAppender

Appender to output the logs in the console is defined as follows:

```
  <!-- attribute: name(Appender Name), target(target of output, "SYSTEM_OUT" or
"SYSTEM_ERR"(default)), follow, ignoreExceptions -->
  <!-- element: Layout(pattern of output), Filters -->
  <Console name="console" target="SYSTEM_OUT">
    <PatternLayout pattern="%d %5p [%c] %m%n" />
  </Console>
```

FileAppender

Appender to output the logs in the file:

```
  <!-- attribute: name(Appender Name), fileName(target file), append(use of append,
true(default) or false), locking, immediateFlush, ignoreExceptions, bufferedIO -->
  <!-- element: Layout(pattern of output), Filters -->
  <!-- append="false” clears the existing log files to log-in fresh off -->
  <File name="file" fileName="./logs/file/sample.log" append="false">
    <PatternLayout pattern="%d %5p [%c] %m%n" />
  </File>
  <File name="mdcFile" fileName="./logs/file/mdcSample.log" append="false">
    <!-- Thread Context Map(also known as MDC) Logs the value that matches the key of the
concerned object - %X{key} -->
    <!-- e.g. ThreadContext.put(“testKey”, “testValue”);, the layout pattern %X{testKey}
applies for“testValue” logging -->
    <PatternLayout pattern="%d %5p [%c] [%X{class} %X{method} %X{testKey}] %m%n"
/>
  </File>
```

RollingFileAppender

RollingFileAppender is used to output the logs in the form of file, according to TriggeringPolicy and RolloverStrategy. With the FileAppender leaves the logs in the target file, you bear a risk of having large files, making it difficult to organize your logs. However, you can replace the target files into the history log when the files exceed the pre-configured limit, in which case the logging is to be performed fresh off.

```
  <!-- attribute: name(Appender Name), fileName(target file), filePattern(history file), append,
immediateFlush, ignoreExceptions, bufferedIO -->
  <!-- element: Layout(Output pattern), Filters, Policy(file rolling conditions), Strategy(file
name and location) -->
  <RollingFile name="rollingFile" fileName="./logs/rolling/rollingSample.log"
filePattern="./logs/rolling/rollingSample.%i.log">
    <PatternLayout pattern="%d %5p [%c] %m%n" />
    <Policies>
      <!-- size unit: Byte(default), KB, MB, or GB -->
      <SizeBasedTriggeringPolicy size="1000" />
    </Policies>
    <!-- The conventional property maxIndex replaced by the element Strategy -->
    <!-- index gains from min(default 1) to max(default 7). See the following example where
max="3” -->
    <!-- fileIndex="min” signifies that the files exceeding 100bytes are to be replaced by the
history log with fileIndex of 1(min) (fixed window strategy) -->
    <!-- When the ensuing files exceed 100bytes, rollingSample.1.log is backed up in
rollingSample.2.log to back the target files up in rollingSample.1.log and commence history
logging fresh off -->
    <DefaultRolloverStrategy max="3" fileIndex="min" />
  </RollingFile>
```

- DailyRollingFileAppender is now replaced by the element <TimeBasedTriggeringPolicy> in RollingFileAppender. You can have the system log whenever you want by way of pre-configuration and pre-conditioning. Use the property Interval to designate the desired rolling interval.

```
  <RollingFile name="rollingFile" fileName="./logs/rolling/dailyRollingSample.log"
filePattern="./logs/daily/dailyRollingSample.log.%d{yyyy-MM-dd-HH-mm-ss}">
    <PatternLayout pattern="%d %5p [%c] %m%n" />
    <Policies>
      <!-- interval(default 1) signifies the rolling is performed per second.-->
      <TimeBasedTriggeringPolicy />
    </Policies>
  </RollingFile>
```

JDBCAppender

JDBCAppender requires that either JNDI DataSource or Connection Factory Method is to be defined to provide the object Connection:

```
  <!-- attribute: name(Appender Name), tableName(RDB Table Name), columnConfigs, filter,
bufferSize, ignoreExceptions, connectionSource -->
  <!-- element: DataSource(jndi datasource information), ConnectionFactory(Connection
Factory information), Column(Table Column Name) -->
  <!-- Table Name logged in the table db_log -->
  <JDBC name="db" tableName="db_log">
    <!-- Define Class and Method to which DB Connection is provided -->
    <!-- JDBCAppender calls the Method EgovConnectionFactory.getDatabaseConnection() --
>
    <ConnectionFactory class="egovframework.rte.fdl.logging.db.EgovConnectionFactory"
method="getDatabaseConnection" />
      <!-- Configure the column where log event is to be inserted. Use PatternLayout to define
the value to be inserted. -->
      <Column name="eventDate" isEventTimestamp="true" />
      <Column name="level" pattern="%p" />
      <Column name="logger" pattern="%c" />
      <Column name="message" pattern="%m" />
      <Column name="exception" pattern="%ex{full}" />
  </JDBC>
```

The conventional Connection Factory is replaced by EgovConnectionFactory in eGovFramework, followed by injection of the dataSource Bean to generate Single Tone. To do so, you need to add the bean Definition as follows:

```
  <bean id="egovConnectionFactory"
class="egovframework.rte.fdl.logging.db.EgovConnectionFactory">
    <property name="dataSource" ref="dataSource" />
  </bean>
```

To use dataSource provided by WAS, replace <ConnectionFactory> for <DataSource jndiName="…" />.

References

Sub-elementas and properties may vary by Appender. Refer to the following manuals for more information:

Log4j 2 Appneders
ConsoleAppender
FileAppender
RollingFileAppender
JDBCAppender

Configuration of Layout

You can designate the format of the log event generated and cause the log output as you desire.
Like Appenders, Layout for Log4j 2 is classified by tag, without use of the Property Class.

Keep in mind the type of Layout, described as follows, vary by output:

Layouts

HTMLLayout **PatternLayout** RFC5424Layout SerializedLayout SyslogLayout XMLLayout

Note that this Section describes the PatternLayout that best fits debugging.

Declaring and Defining PatternLayout

PatternLayout is defined as the sub-elements of the element Appender.

```
  <Console>
    <!-- Applying the default pattern, "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}
- %msg%n" -->
    <PatternLayout />
  </Console>
```

When <PatternLayout /> is declared, the default pattern is applied after which date, time, class, logger name, message and other information can be used to comprise the desired log messages.

PatternLayout

PatternLayout starts with %, followed by format modifiers and conversion character.
e.g. %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n

Pattern	Description
c, logger	Causes the name of the selected Logger output to bring about a logging event

C, class	Causes the name of Class output to where a logging event arose
M, method	Causes the name of Method output to where a logging event arose
F, file	Causes the file name of Class output to where a logging event arose
l, location	Causes the full name and method (file name: Line #) output to where a logging event arose
d, date	Causes the date and time of a logging event output to where a logging event arose, abiding by the pattern pre-defined in the class \\SimpleDateFormat
L, line	Causes Line # output to where a logging event arose
m, msg, message	Causes the message sent out from the log output
n	New Line
p, level	Causes the level of logging event output
r, relative	Process time (milliseconds)
t, thread	Causes the name of thread giving rise to the logging event output
%%	Pattern used to display %

References

Log4j 2 Layouts
PattenLayout

References

Apache Logging Services Project