**Lesson 19: Logging with Log4J**

* **What is a Logger configuration file? How many ways you can implement (XML and property file)?**

**1)** The Java Logging API can be configured in two ways:

1. **Via a configuration class.**
2. **Via a configuration file.**

The initialization of the configuration is taken care of by the java.util.logging.LogManager class.

**Configuration Class**

You can use a Java class to configure the Java Logging API. You do so by specifying the name of the class in the JVM parameter java.util.logging.config.class. It is the constructor of that class that should load the configuration and apply it to the Logger's in the hierarchy.

**Configuration File**

If no configuration class is specified, one can instead specify a configuration file (but no configuration class can be specified then).

The Java Logging API has a default logging configuration file located at "lib/logging.properties", inside the JRE directory. If one can edit this file, one needs to edit the default logging settings for the entire JRE, for every program executed. This is most often not what one wants to do, though.

Instead, one can set a separate configuration file for their application. It is done so by setting the JVM propertyjava.util.logging.config.file to point to this file.

The configuration file is a standard property file from Java. Inside this property file one can set properties that configure the various Logger's and Handler's used in your application.

**2)** There are three ways in which one can implement XML and property file. They are

**a) By XML-Based Configuration**

**b) By Java Based Configuration**

**c) By Annotation Based Configuration**

* **Log4J API Components-Discuss**

The main components are:

1. **Logger**
2. **Appender**
3. **Layout**
4. **Logger**

Logger is a class, in org.apache.log4j.One needs to create Logger object one per java class. This component enables Log4j in our java class. Logger methods are used to generate log statements in a java class instead of sopls. So in order to get an object of Logger class, one needs to call a static factory method [factory method will gives an object as return type]

One must create Logger object right after our class name, like

**Example:**

public class MediaComposer {

   static Logger l = Logger.getLogger(MediaComposer.class.getName ());

   public static void main(String [] args) {

     // logic will go here

   }

}

Logger object has some methods, actually we used to print the status of our application by using these methods only

We have totally 5 methods in Logger class

* **debug()**
* **info()**
* **warn()**
* **error()**
* **fatal()**

1. **Appender**

Appender job is to write the messages into the external file or database or smtp. Logger classes generates some statements under different levels right, this Appender takes these log statements and stores in some files or database. Appender is an interface. In log4j we have different Appender implementation classes:

* **FileAppender** [ For writing into a file ]
* **ConsoleAppender** [For Writing into console ]
* **JDBCAppender** [ For Databases ]
* **SMTPAppender** [ For Mails ]
* **SocketAppender** [ For remote storage ]
* **SocketHubAppender**
* **SyslogAppendersends**
* **TelnetAppender**

Again in FileAppender we have 2 more

* **RollingFileAppender**
* **DailyRollingFileAppender**

1. **Layout**

This component specifies the format in which the log statements are written into the destination repository by the appender.

We have different type of layout classes in log4j:

* **SimpleLayout**
* **PatternLayout**
* **HTMLLayout**
* **XMLLayout**
* **Log4j levels-Discuss**

1. **ALL-** All levels including custom levels
2. **DEBUG-** Designates fine-grained informational events that are most useful to debug an application
3. **ERROR-** Designates error events that might still allow the application to continue running.
4. **FATAL-** Designates very severe error events that will presumably lead the application to abort.
5. **INFO-** Designates informational messages that highlight the progress of the application at coarse-grained level.
6. **OFF-** The highest possible rank and is intended to turn off logging
7. **TRACE-** Designates finer-grained informational events than the DEBUG
8. **WARN-** Designates potentially harmful situations.

## How do Levels Works?

A log request of level p in a logger with level q is enabled if p >= q. This rule is at the heart of log4j. It assumes that levels are ordered. For the standard levels, we have ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.

The Following example shows how we can filter all our DEBUG and INFO messages. This program uses of logger method setLevel(Level.X) to set a desired logging level:

This example would print all the messages except Debug and Info:

public class MediaComposer {

private static org.apache.log4j.Logger log = Logger.getLogger(MediaComposer.class);

public static void main(String [] args)

{

log.setLevel(Level.WARN);

log.trace(“Trace Message!”);

log.debug(“Debug Message!”);

log.info(“Info Message!”);

log.warn(“Warn Message!”);

log.error(“Error Message!”);

log.fatal(“Fatal Message!”);

}

}

* **What is Appender Logger, root logger?**

Logging requests can be sent to multiple destinations, such as files, databases, syslog and others. Such destinations are called appenders. Appenders are attached to [loggers](https://logging.apache.org/log4php/docs/loggers.html) and each logger can have multiple attached appenders.

1. LoggerAppenderConsole- Console, directly to the stdout or stderr stream
2. LoggerAppenderEcho- Console, Using the PHP echo command
3. LoggerAppenderFile – A file
4. LoggerAppenderDailyFile- A file (new file each day).
5. LoggerAppenderRollingFile - A file (new file when a specified size has been reached).
6. LoggerAppenderMail- Sends the log via email. The entire log is sent in one email.
7. LoggingAppenderPDO - Database
8. LoggerAppenderSocekt – A network socket. ..etc.

**An appender has the following properties**:

* A name which uniquely identifies it, in this case default.
* A class which specifies which appender class will be used to handle the requests. Since we wish to log to a file, LoggerAppenderFile is used in this case.
* A layout which transforms the logging events to string which can be logged. A layout is required by most appenders, but some do not require it, such as the database appender. If a layout is not defined, the appenders will use a default layout.
* Zero or more parameters which configure the appender behaviour. In this example, the file parameter governs the path to the file which will be used for logging, and append defines that log messages should be appended to the file, instead of truncating it.

### Linking appenders to loggers

A logger can be linked to one or more appenders. Also, multiple loggers can share the same appender.

**Consider the following configuration:**

<log4php: configuration xmlns:log4php="http://logging.apache.org/log4php/">

<appender name="primus" class="LoggerAppenderConsole" />

<appender name="secundus" class="LoggerAppenderFile">

<param name="file" value="/var/log/my.log" />

</appender>

<logger name="main">

<appender\_ref ref="primus" />

<appender\_ref ref="secundus" />

</logger>

<logger name="alternative">

<appender\_ref ref="primus" />

</logger>

</log4php: configuration>

This configures two appenders, called primus and secundus, and two loggers named main and alternative. The logger main is linked to primus and secundus and will therefore forward logging events to both of them. In other words, it will log both to console and to a file. Logger alternative is only linked to appender primus and will therefore only log to the console.

* **Logger Configuration File Format**

The logging configuration can be initialized using a logging configuration file that will be read at startup. This logging configuration file is in standard java.util.Properties format.

Alternatively, the logging configuration can be initialized by specifying a class that can be used for reading initialization properties. This mechanism allows configuration data to be read from arbitrary sources, such as LDAP, JDBC, etc. See the [Log Manager API Specification](https://docs.oracle.com/javase/8/docs/api/java/util/logging/LogManager.html) for details.

There is a small set of global configuration information. This is specified in the description of the Log Manager class and includes a list of root-level Handlers to install during startup.

The initial configuration may specify levels for particular loggers. These levels are applied to the named logger and any loggers below it in the naming hierarchy. The levels are applied in the order they are defined in the configuration file.

The initial configuration may contain arbitrary properties for use by Handlers or by subsystems doing logging. By convention these properties should use names starting with the name of the handler class or the name of the main Logger for the subsystem.

For example, the Memory Handler uses a property "java.util.logging.MemoryHandler.size" to determine the default size for its ring buffer.

### **Default Configuration**

The default logging configuration that ships with the JRE is only a default, and can be overridden by ISVs, system admins, and end users.

The default configuration makes only limited use of disk space. It doesn't flood the user with information, but does make sure to always capture key failure information.

The default configuration establishes a single handler on the root logger for sending output to the console.

### **Dynamic Configuration Updates**

Programmers can update the logging configuration at run time in a variety of ways:

* FileHandlers, MemoryHandlers, and PrintHandlers can all be created with various attributes.
* New Handlers can be added and old ones removed.
* New Loggers can be created and can be supplied with specific Handlers.
* Levels can be set on target Handlers.