LOG4J TUTORIAL

Apache Log4j Logging Framework
Tutorial with Example Projects

TABLE OF CONTENTS

LOG4J TUTORIAL

Pankaj Kumar

Log4j is most used logging framework in java applications. In this **log4j tutorial**, we will go through log4j basics, it's configuration and then see it in action in java standalone application and then in java web application.

Log4j Jars

There are many ways to get log4j jars.

1. My favourite is to get all the dependencies through Maven. You can add below dependencies in your pom.xml file to get log4j jar. <dependency>
   <groupId>log4j</groupId>
   <artifactId>log4j</artifactId>
   <version>1.2.17</version>
   </dependency>
2. If you are using gradle, then add dependency as: `'log4j:log4j:1.2.17'`
3. If you are not using any build tools, you can download the log4j jar from Apache Log4j Official page and include into your project classpath.

Log4j Configuration

Log4j supports both properties based as well as XML based configuration. Here I will use XML based configuration, you can see log4j properties based configuration in the linked post.

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

debug="false">

<!-- console appender -->

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
<param name="Target" value="System.out" />
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%-5p %c1 - %m%n" />
</layout>
</appender>

<!-- rolling file appender -->
<appender name="file" class="org.apache.log4j.RollingFileAppender">
<param name="File" value="logs/main.log" />
<param name="Append" value="true" />
<param name="ImmediateFlush" value="true" />
<param name="MaxFileSize" value="10MB" />
<param name="MaxBackupIndex" value="5" />

<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%d %dZ %-5p (%F:%L) -
%m%n" />
</layout>
</appender>

<logger name="com.journaldev.log4j" additivity="false">
<level value="DEBUG" />
<appender-ref ref="file" />
<appender-ref ref="console" />
</logger>

<logger name="com.journaldev.log4j.logic" additivity="false">
<level value="INFO" />
<appender-ref ref="file" />
</logger>

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

</log4j:configuration>
```

The most important part of log4j configuration files are:

- **Appenders**: Here we define the logging strategy, such as which type of appender class to use, for example `org.apache.log4j.ConsoleAppender` or `org.apache.log4j.RollingFileAppender`. `org.apache.log4j.PatternLayout` is used to define the logging pattern. `%d %dZ  %-5p (%F:%L) - %m%n` will append time, thread, logging level, class name with line number. We can define any number of appenders in our log4j configuration file.
- **logger**: It's used to provide mapping between java packages/classes with appenders and logging levels. As you can see, multiple appenders can be used with a single logger. If multiple logger matches with the java class package, more specific one is used.
- **root**: This is used when there is no logger defined for java packages. For example, if have some classes in `com.journaldev.beans` package then it won't match with any logger. So root logger will be used for logging configurations.

Once log4j configuration file is ready, we have to configure it in the code before we can actually start logging. Below is the way to do that.

```
//for XML based configuration
DOMConfigurator.configure("log4j.xml");
//for properties based configuration
PropertyConfigurator.configure("log4j.properties");
```

Note that if our log4j configuration file name is `log4j.xml` or `log4j.properties`, then we can skip this step because log4j tries to automatically load these files from classpath.

Log4j usage

Now let's see how they can be used in the code. We have created two classes for our above defined loggers.

```
package com.journaldev.log4j.main;

import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;

import com.journaldev.log4j.logic.MathUtils;
```

```
public class Log4jExample

static
init();

private final static Logger logger = Logger.getLogger(Log4jExample.class);

public static void main(String args)

logger.debug("My Debug Log");
logger.info("My Info Log");
logger.warn("My Warn Log");
logger.error("My error log");
logger.fatal("My fatal log");

MathUtils.add(4,5);
MathUtils.add(40,50);
MathUtils.add(1,5);

/**
* method to init log4j configurations
*/
private static void init()
DOMConfigurator.configure("log4j.xml");
```

We can create a final Logger instance once and then reuse it across the class. Notice that I am configuring log4j in `init()` method that is called at the time of class loading in static block.

```
package com.journaldev.log4j.logic;

import org.apache.log4j.Logger;

public class MathUtils

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

public static int add(int x, int y)
logger.debug("inputs are:"+x+", "+y);
return x+y;
```

Now when we run above main method, we will get below logs in main.log file.

```
2016-05-12 21:22:44,610 +0530 DEBUG (Log4jExample.java:18) - My Debug Log
2016-05-12 21:22:44,611 +0530 INFO (Log4jExample.java:19) - My Info Log
2016-05-12 21:22:44,611 +0530 WARN (Log4jExample.java:20) - My Warn Log
2016-05-12 21:22:44,612 +0530 ERROR (Log4jExample.java:21) - My error log
2016-05-12 21:22:44,612 +0530 FATAL (Log4jExample.java:22) - My fatal log
```

You will also see logs printing into console.

```
DEBUG Log4jExample - My Debug Log
INFO Log4jExample - My Info Log
WARN Log4jExample - My Warn Log
ERROR Log4jExample - My error log
FATAL Log4jExample - My fatal log
```

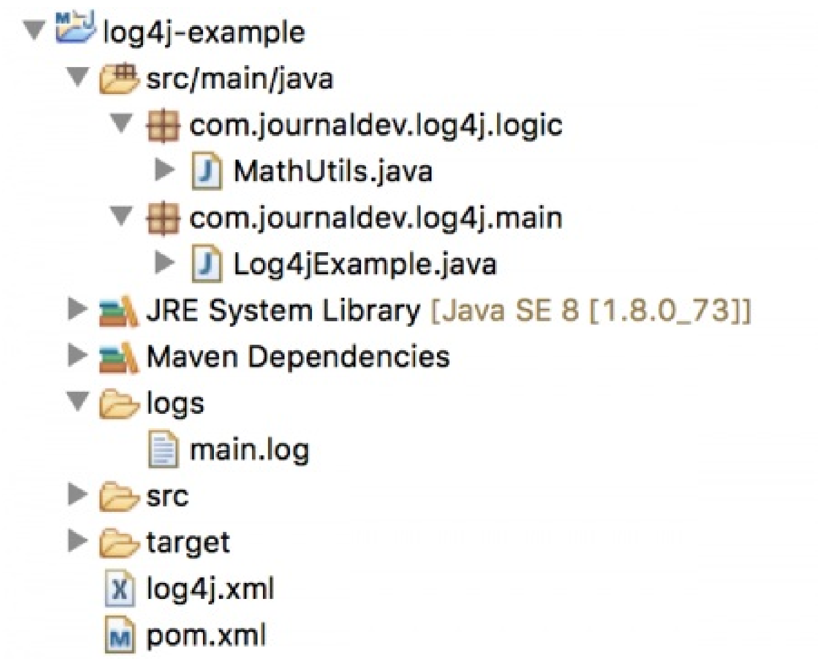
Notice that there are no logs getting printed from MathUtils class, it's because it's logger level is INFO that is higher than DEBUG. Just change it's logging level to DEBUG like below.

```
<logger name="com.journaldev.log4j.logic" additivity="false">
<level value="DEBUG" />
<appender-ref ref="file" />
</logger>
```

Now when you will run the program again, main.log file will have below logs too.

```
2016-05-12 21:23:56,151 +0530 DEBUG (MathUtils.java:10) - inputs are:4, 5
2016-05-12 21:23:56,151 +0530 DEBUG (MathUtils.java:10) - inputs are:40, 50
2016-05-12 21:23:56,153 +0530 DEBUG (MathUtils.java:10) - inputs are:1, 5
```

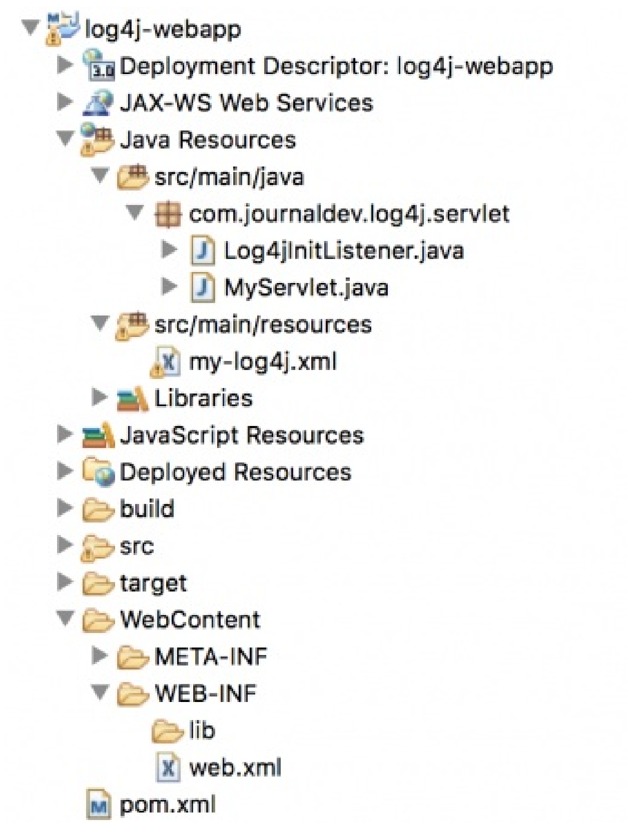
That's it for using log4j in java standalone application. Below image shows the final project structure in eclipse.



```
log4j-example
    src/main/java
        com.journaldev.log4j.logic
            MathUtils.java
        com.journaldev.log4j.main
            Log4jExample.java
    JRE System Library [Java SE 8 [1.8.0_73]]
    Maven Dependencies
    logs
        main.log
    src
    target
    log4j.xml
    pom.xml
```

Log4j in Java Web Application

Now let's see how to use log4j in java web application. Create a "Dynamic Web Project" and then convert it to Maven. Below image shows the final structure of the project.



```
log4j-webapp
  Deployment Descriptor: log4j-webapp
  JAX-WS Web Services
  Java Resources
    src/main/java
      com.journaldev.log4j.servlet
        Log4jInitListener.java
        MyServlet.java
    src/main/resources
      my-log4j.xml
    Libraries
  JavaScript Resources
  Deployed Resources
  build
  src
  target
  WebContent
    META-INF
    WEB-INF
      lib
      web.xml
  pom.xml
```

Below is the log4j configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">


debug="false">

<!-- console appender -->
<appender name="console" class="org.apache.log4j.ConsoleAppender">
<param name="Target" value="System.out" />
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%-5p %c1 - %m%n" />
</layout>
</appender>

<!-- rolling file appender -->
<appender name="file" class="org.apache.log4j.RollingFileAppender">
<param name="File" value="$catalina.home/logs/main.log" />
<param name="Append" value="true" />
<param name="ImmediateFlush" value="true" />
<param name="MaxFileSize" value="10MB" />
<param name="MaxBackupIndex" value="5" />
```

```
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%d %dZ %-5p (%F:%L) -
%m%n" />
</layout>
</appender>

<logger name="com.journaldev.log4j" additivity="false">
<level value="DEBUG" />
<appender-ref ref="file" />
<appender-ref ref="console" />
</logger>

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

</log4j:configuration>
```

More of less it's similar to earlier configuration, except that we are injecting `catalina.home` variable to generate log files into tomcat logs directory.

Since we have to configure log4j before it's being used, we can load it by defining a `ServletContextListener` as below.

```
package com.journaldev.log4j.servlet;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

import org.apache.log4j.xml.DOMConfigurator;

@WebListener
public final class Log4jInitListener implements ServletContextListener

public Log4jInitListener()

public void contextDestroyed(ServletContextEvent
paramServletContextEvent)
```

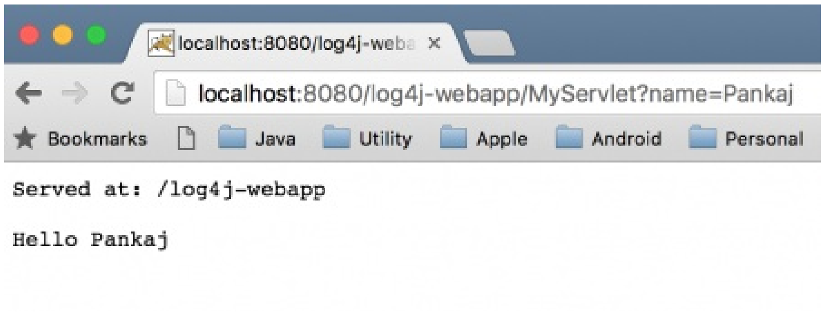
```
public void contextInitialized(ServletContextEvent servletContext)
String webAppPath = servletContext.getServletContext().getRealPath("/");
String log4jFilePath = webAppPath + "WEB-INF/classes/my-log4j.xml";
DOMConfigurator.configure(log4jFilePath);
System.out.println("initialized log4j configuration from file:"+log4jFilePath);
```

Notice the use of `ServletContext` to get the full path of log4j configuration file. This is something additional we have to do because we don't know the full path of the log4j file at runtime and we don't want to hardcode it either.

Below is a simple Servlet class using Logger to log some messages.

```
package com.journaldev.log4j.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.log4j.Logger;

@WebServlet("/MyServlet")
public class MyServlet extends HttpServlet

private static final long serialVersionUID = 1L;

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

public MyServlet()

protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
String name = request.getParameter("name");
logger.info("Name parameter value = "+name);

PrintWriter out = response.getWriter();

out.append("Served at: ").append(request.getContextPath());
out.append("\n\nHello "+name);
out.flush();
```

Just export the project as WAR file and then deploy into Tomcat server, below image shows when we call the servlet in browser.


localhost:8080/log4j-weba
localhost:8080/log4j-webapp/MyServlet?name=Pankaj
Bookmarks
Java
Utility
Apple
Android
Personal

```
Served at: /log4j-webapp

Hello Pankaj
```

You will get below kind of logs in tomcat logs directory main.log file.

```
pankaj:logs pankaj$ tail -f main.log
2016-05-12 21:46:33,038 +0530 INFO (MyServlet.java:29) - Name parameter value = Pankaj
```

Since we are also logging to console that goes to catalina.out file, you will find below logs in catalina log file.

```
pankaj:logs pankaj$ tail -f catalina.out
INFO MyServlet - Name parameter value = Pankaj
```

That's it for a quick tutorial on log4j.

References:

- Log4j Official Website
- Log4j on Wikipedia
- Log4j initialization by implementing LifecycleListener

LOG4J LEVELS

Pankaj Kumar

You will notice that there are many methods to log messages in log4j. For example:

```
logger.trace("My Log message");
logger.debug("My Log message");
logger.info("My Log message");
```

Actually they corresponds to **log4j levels.**

Log4j Levels

Log4j provides many logging levels. Below is the complete list.

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

ALL and OFF are special logging levels and should be used in extreme situations. I have never used these personally at any point of time.

Log4j Level Order/Priority

Trace is of the lowest priority and Fatal is having highest priority. Below is the log4j logging level order.

Trace < Debug < Info < Warn < Error < Fatal.

When we define logger level, anything having higher priority logs are also getting printed. For example, if logger level is INFO then debug logs will not be printed but Warn logs will be printed because of higher priority.

Log4j Filters

Let's say we want to log only INFO and FATAL events but not WARN and ERROR events. In these scenarios, we can take help of log4j Filters. We can extend `org.apache.log4j.spi.Filter` class and implements it's `decide(LoggingEvent event)` method to provide custom filtering capabilities.

```
package com.journaldev.log4j.filters;

import org.apache.log4j.Level;
import org.apache.log4j.spi.Filter;
import org.apache.log4j.spi.LoggingEvent;

public class MyLog4jFilter extends Filter

/**
* My custom filter to only log INFO and FATAL events
*/
@Override
public int decide(LoggingEvent event)
if(event.getLevel() == Level.INFO || event.getLevel() == Level.FATAL)
return ACCEPT;
else return DENY;
```

Above custom Filter will log only INFO and FATAL events, below is the XML log4j configuration for this.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
```

```
debug="false">

<!-- console appender -->
<appender name="console" class="org.apache.log4j.ConsoleAppender">
<param name="Target" value="System.out" />
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%-5p %c1 - %m%n" />
</layout>
<filter class="com.journaldev.log4j.filters.MyLog4jFilter" />
</appender>

<logger name="com.journaldev.log4j" additivity="false">
<level value="TRACE" />
<appender-ref ref="console" />
</logger>

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

</log4j:configuration>
```

Notice the usage of Filter class in the console appender. Below is a simple class doing basic logging.

```
package com.journaldev.log4j.main;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.xml.DOMConfigurator;

public class Log4jExample

static
init();

private final static Logger logger = Logger.getLogger(Log4jExample.class);

public static void main(String args)
```

```
logger.trace("My Trace Log");
logger.debug("My Debug Log");
logger.info("My Info Log");
logger.warn("My Warn Log");
logger.error("My error log");
logger.fatal("My fatal log");

/**
* method to init log4j configurations
*/
private static void init()
DOMConfigurator.configure("log4j.xml");
```

On running this program, it generates below logs into console.

```
INFO Log4jExample - My Info Log
FATAL Log4jExample - My fatal log
```

We can do even more complex filtering by creating our own custom filter classes. Notice that for this particular case, we can use `org.apache.log4j.varia.LevelMatchFilter` and `org.apache.log4j.varia.DenyAllFilter` classes as shown in below appender.

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
<param name="Target" value="System.out" />
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%-5p %c1 - %m%n" />
</layout>
<filter class="org.apache.log4j.varia.LevelMatchFilter">
<param name="LevelToMatch" value="INFO" />
<param name="AcceptOnMatch" value="true" />
</filter>
<filter class="org.apache.log4j.varia.LevelMatchFilter">
<param name="LevelToMatch" value="FATAL" />
<param name="AcceptOnMatch" value="true" />
</filter>
<filter class="org.apache.log4j.varia.DenyAllFilter"/>
</appender>
```

We also have `org.apache.log4j.varia.LevelRangeFilter` class that can be used to reject messages with priorities outside a certain range.

That's all for a quick understanding of log4j levels and filters.

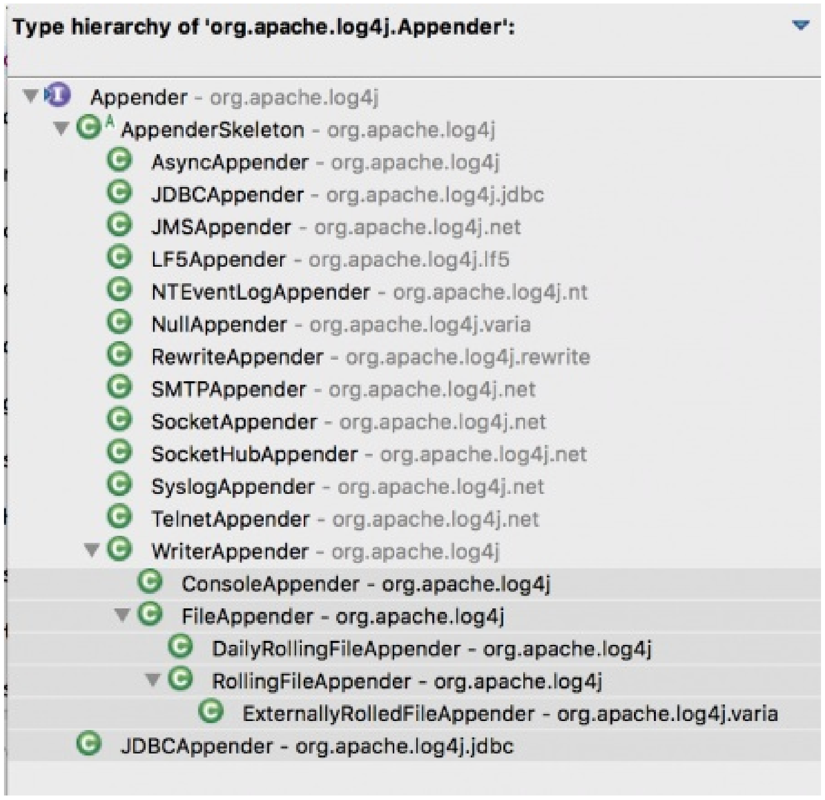
References:

- Log4j Level API
- Log4j Custom Filter StackOverflow

LOG4J APPENDERS

Pankaj Kumar

Log4j Appenders provides configuration for logging such as console, file, database etc. Below image shows the log4j Appender class hierarchy.



```
Type hierarchy of 'org.apache.log4j.Appender':
Appender - org.apache.log4j
  AppenderSkeleton - org.apache.log4j
    AsyncAppender - org.apache.log4j
    JDBCAppender - org.apache.log4j.jdbc
    JMSAppender - org.apache.log4j.net
    LF5Appender - org.apache.log4j.lf5
    NTEventLogAppender - org.apache.log4j.nt
    NullAppender - org.apache.log4j.varia
    RewriteAppender - org.apache.log4j.rewrite
    SMTPAppender - org.apache.log4j.net
    SocketAppender - org.apache.log4j.net
    SocketHubAppender - org.apache.log4j.net
    SyslogAppender - org.apache.log4j.net
    TelnetAppender - org.apache.log4j.net
    WriterAppender - org.apache.log4j
      ConsoleAppender - org.apache.log4j
      FileAppender - org.apache.log4j
        DailyRollingFileAppender - org.apache.log4j
        RollingFileAppender - org.apache.log4j
          ExternallyRolledFileAppender - org.apache.log4j.varia
  JDBCAppender - org.apache.log4j.jdbc
```

log4j Appender

This is the base of log4j Appenders that defines the methods to implement by an appender.

log4j AppenderSkeleton

This class provides the code for common functionality, such as support for threshold filtering and support for general filters. This is the base implementation that is extended by all other appenders such as `JDBCAppender` , `AsyncAppender` , `ConsoleAppender` etc. It has only one abstract method `append(LoggingEvent event)` that is required to be implemented by all appenders. We can write our own custom appender by implementing this method.

Commonly used log4j Appenders

Some of the most commonly used appenders are:

- **ConsoleAppender**: ConsoleAppender appends log events to System.out or System.err using a layout specified by the user. The default target is System.out. It's good for debugging purposes, but not much beneficial to use in production environment.
- **RollingFileAppender**, **DailyRollingFileAppender**: These are the most widely used appenders that provide support to write logs to file. `RollingFileAppender` is used to limit the log file size and number of backup files to keep. `DailyRollingFileAppender` is used to log into files on date basis. However DailyRollingFileAppender has been observed to exhibit synchronization issues and data loss and not recommended to use.
- **JDBCAppender**: The JDBCAppender provides for sending log events to a database. Each append call adds to an ArrayList buffer. When the buffer is filled each log event is placed in a sql statement (configurable) and executed. BufferSize, db URL, User, & Password are configurable options in the standard log4j ways.
- **AsyncAppender**: The AsyncAppender lets users log events asynchronously. The AsyncAppender will collect the events sent to it and then dispatch them to all the appenders that are attached to it. You can attach multiple appenders to an AsyncAppender. Note that we can configure it only through XML based i.e `DOMConfigurator`. This is useful when you are generating a lot of logs and don't care if they are being logged instantly. There are chances of logs getting lost incase server crash. The AsyncAppender uses a separate thread to serve the events in its buffer.
- **JMSAppender**: A simple appender that publishes events to a JMS Topic. The events are serialized and transmitted as JMS message type ObjectMessage.

Log4j Appenders XML Configuration

Below is the XML based configuration of commonly used ConsoleAppender and RollingFileAppender classes.

```
<!-- console appender -->
<appender name="console" class="org.apache.log4j.ConsoleAppender">
<param name="Target" value="System.out" />
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%-5p %c1 - %m%n" />
</layout>
</appender>

<!-- rolling file appender -->
<appender name="file" class="org.apache.log4j.RollingFileAppender">
<param name="File" value="logs/main.log" />
<param name="Append" value="true" />
<param name="ImmediateFlush" value="true" />
<param name="MaxFileSize" value="10MB" />
<param name="MaxBackupIndex" value="5" />

<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%d %dZ %-5p (%F:%L) -
%m%n" />
</layout>
</appender>
```

You can check the appender classes code to find out the parameters you can configure. For example in JDBCAppender you can configure databaseURL, databaseUser, databasePassword etc.

Log4j Appender Properties Configuration

A simple example showing appenders configuration through property file. It's defining all the above xml based configuration.

```
#Define console appender
log4j.appender.console=org.apache.log4j.ConsoleAppender
logrj.appender.console.Target=System.out
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-5p %c1 - %m%n

#Define rolling file appender
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=logs/main.log
```

```
log4j.appender.file.Append=true
log4j.appender.file.ImmediateFlush=true
log4j.appender.file.MaxFileSize=10MB
log4j.appender.file.MaxBackupIndex=5
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d %dZ %-5p (%F:%L) -
%m%n
```

That's all for a quick roundup on log4j appenders.