**INTRODUCTION TO Log4J**

**Introduction**

Any Real Time Java Application contains many components(projects) with different package structure with N number of classes,interfaces,jsp files etc.

Although developer tries their level best to write robust code, Real Time Applications are not guaranteed to be 100% bug free.Most often we have also encountered bug in live environment.

If We talk about live environment,then tracing bugs in live environment must be quick else we may lose customer and so the business.

**RealTime UseCase**

It was a vacation period and I was just trying to book my ticket from Hyderabad to Bhubaneswar and in my CITIBANK payment page I got Some Exception,I tried again for a couple of time but still got the same Exception.Then I had to book my ticket from HDFC Account and it went good

My booking time was around 12:00 AM and in this period generally the Application undergoes maintance window. Which means certain patches are installed and few things are re-configured if needed.That may not be the only reason of the Exception? Sometimes few services may go down due to some failure.

Certainly CITIBANK might have lost many customer at that time until it was fixed by support team.

If you are working on support,then you will have to deal with these kind of situation.You are not the actual programmer of the Application and You need to find the root cause of the Problem and also need to provide atleast some temporary solution to keep it running. You will not have access to Server console

So the question is,How to start analyzing the problem.

So we need some mechanism to write the exception outputs to a file(log file),so that we can analyse the log file later which can help us identifying and resolving the problem.

**Introduction to Logging:**

Logging is a technique or process to record various information to any output device such as files,outputScreens etc.

* Aeroplanes logs about 2.5 GB of Data each 30 minutes to its black box,so that those data can be read later to study any failure if it comes on the flight.
* All Software Programs writes the Exception,Error,Information data to a file(log file),so that it can be used to understand the flow of program or to debug the application

**Why Logging is needed:-**

* To understand the flow of the application.
* To log Exceptions occurred in the application.
* To log some Application information to the file.

**Advantage of Logging Framework**

***Problem diagnosis***: No matter how well written our code is, there may be some problems hidden in it. As soon as the triggering conditions occur, the hidden problems come to the surface. If our applications have well-written code for logging the internal state of the system, we will be able to detect the problems precisely and quickly.

***Quick debugging*:** Once we diagnose the problem, we know exactly how to solve the problem. The logging trace should be aimed at showing the precise location of the problem,which means we will be able to debug the application in less time. Well-planned and well-written logging code greatly reduces the overall cost of debugging the application.

***Easy maintenance*:** Applications with a good logging feature are easy to debug and therefore easy to maintain compared to any application without a similar logging feature. The logging information typically contains more information than the debugging trace.

***History*:** A good logging feature in an application results in logging information being preserved in a structured way at a desired location. The location may be a file, database, or remote machine. All this enables system a future date by going through the logging history.

***Cost and time savings*:** As explained, well-written logging code offers quick debugging,easy maintenance, and structured storage of an application’s runtime information. This makes installation, day-to-day maintenance, and debugging much more cost- and time-effective.

**Disadvantage of Logging Framework**

* Logging adds runtime overhead due to the generation of logging information and the device Input/Output (I/O) related to publishing logging information. Log4j needs 5 nanoseconds each time to log a statement.
* Logging adds programming overhead due to the extra code required for producing logging information. The logging process increases the size of the code.
* Badly produced logging information can cause confusion.
* Badly written logging code can seriously affect the application’s performance.
* Last but not least, logging requires planning ahead, as adding logging code at a late stage of development is difficult.

**Popular Java-Based Logging APIs**

Once logging concepts were proven successful, they were put into use as generic logging APIs. A few Java-based logging APIs are available in the market. Some of them are proprietary, and some are open source. Out of all the available APIs, the following are most popular in the Java community.

**JDK Logging API**

JDK has its own logging API in its java.util.logging package. This API originated from the JSR 47.

The JDK logging API is, in essence, a scaled-down version of Apache log4j (discussed in the next section). The logging concepts captured in this API involve logging levels and different logging destinations and formats. The JDK logging API is well suited for simple applications with simple logging requirements. Despite a few limitations, this API provides all the basic features that you need to produce effective logging information.

**Apache log4j**

Apache log4j is an open-source logging API. This API, which evolved from the E.U. SEMPER (Secure Electronic Marketplace for Europe) project, is a popular logging package in Java. It allows great control over the granularity of logging statements. A main benefit of this API is that it is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great

variety of destinations, such as a database, file, console, Windows NT event log, UNIX Syslog, Java Message Service JMS), and so on. It also allows application developers to choose from various formatting styles, such as XML, HTML, etc. Overall, log4j is a feature-rich, well-designed extendible logging framework, and provides more capabilities than the JDK logging API. For example, the configuration of log4j is much more flexible than that of the JDK logging API. The JDK logging API can be configured only through a properties-style configuration file, but log4j supports both properties- and XML-style configuration.

**Commons Logging API**

The Commons logging API is another logging effort from Apache. The goal of this API is to provide a seamless transition from one logging API to another. Depending upon the presence of a logging framework in the classpath, the Commons logging API will try to use the available API to carry out application logging. The Commons logging API runs its own discovery process to find out which logging API is available in the classpath. It tends to provide the lowest common denominator of any two logging APIs. For example, between log4j and the JDK logging API, it will provide a seamless transition for the features common in both—so we would miss any extra features used in log4j implementation is a highly scalable, robust, and versatile logging framework.This API simplifies the writing of logging code within an application, yet allows the flexibility of controlling logging activity from an external configuration file. It also allows us to publish logging information to desired granularity depending on the detail of the logging information suitable to each application.

Apache log4j is also capable of publishing logging information to various destinations such as files, consoles, and NT event logs. Moreover, logging information can even be distributed over Java Message Service ( JMS) or Java Database Connectivity (JDBC), or can be output to a TCP/IP socket. This API lets us take logging information and publish or print it in different formats and layouts that are human-readable and reusable by any error-handling and analyzer program

**Log4J Framework Features:**

* log4j is optimized for speed.
* log4j is based on a named logger hierarchy.
* log4j is fail-stop but not reliable.
* log4j is thread-safe.
* log4j is not restricted to a predefined set of facilities.
* Logging behavior can be set at runtime using a configuration file. Configuration files can be property files or in XML format.
* log4j is designed to handle Java Exceptions from the start.
* log4j can direct its output to a file, the console, an java.io.OutputStream, java.io.Writer, a remote server using TCP, a remote Unix Syslog daemon, to a remote listener using JMS, to the NT EventLog or even send e-mail.
* log4j uses 5 levels, namely DEBUG, INFO, WARN, ERROR and FATAL.
* The format of the log output can be easily changed by extending the Layout class.
* The target of the log output as well as the writing strategy can be altered by implementations of the Appender interface.
* log4j supports multiple output appenders per logger.
* log4j supports internationalization.

**Installing log4j**

Apache log4j is an open-source project from Apache. You must meet the following criteria to successfully install and use log4j:

 Get the latest version of the log4j binary distribution from [http://jakarta.apache.org/log4j.We will use 1.2](http://jakarta.apache.org/log4j.We%20will%20use%201.2) version. So download them from below link. <https://logging.apache.org/log4j/1.2/download.html>

Direct Link: [**http://mirror.cc.columbia.edu/pub/software/apache/logging/log4j/1.2.17/log4j-1.2.17.zip**](http://mirror.cc.columbia.edu/pub/software/apache/logging/log4j/1.2.17/log4j-1.2.17.zip)

Add log4j-1.2.17.jar to your eclipse build path

**Overview of the log4j Architecture**

The architecture of the log4j API is layered. Each layer consists of different objects performing different tasks. The top layer captures the logging information, the middle layer is involved in analyzing and authorizing the logging information, and the bottom layer is responsible for formatting and publishing the logging information to a destination. In essence, log4j consists of three types of primary objects:

* **Logger:** The Logger object (known as the Category object in releases prior to log4j 1.2) is responsible for capturing logging information. Logger objects are stored in a namespace hierarchy.
* **Appender:** The Appender object is responsible for publishing logging information to various preferred destinations. Each Appender object will have at least one target destination attached to it. For example, a ConsoleAppender object is capable of printing logging information to a console.
* **Layout:** The Layout object is used to format logging information in different styles. Appender objects utilize Layout objects before publishing logging information. Layout objects play an important role in publishing logging information in a way that is human-readable and reusable.

The preceding core objects are central to the architecture of log4j. Apart from them, there are several auxiliary objects that can plug and play to any layer of the API. These objects help manage the different Logger objects active within an application and fine tune the logging process.

Next, let’s go over the principal auxiliary components in the log4j framework that play a vital role in the logging framework:

* **Level:** The Level object, previously referred to as the Priority object, defines the granularity and priority of any logging information. Each piece of logging information is accompanied by its appropriate Level object, which tells the Logger object about the priority of the information. There are seven levels of logging defined within the API: OFF, DEBUG, INFO, ERROR, WARN, FATAL, and ALL. Each level has a unique integer value. The Level values can be arranged in an ascending manner:

ALL<DEBUG<INFO<WARN<ERROR<FATAL<OFF

where ALL means most of the logging information will be published and OFF means none of the logging information will be published.

**Levels in Log4j**

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

* **Filter:** The Filter object is used to analyze logging information and make further decisions on whether that information should be logged or not. In the log4j context, Appender objects can have several Filter objects associated with them. If logging information is passed to a particular Appender object, all the Filter objects associated with that Appender need to approve the logging information before it can be published to the preferred destination attached to the Appender. Filter objects are very helpful in filtering out unwanted logging information based on any application-specific criteria.
* **ObjectRenderer:** The ObjectRenderer object is specialized in providing a String representation of different objects passed to the logging framework. More precisely, when the application passes a custom Object to the logging framework, the logging framework will use the corresponding ObjectRenderer to obtain a String representation of the passed Object.This is used by Layout objects to prepare the final logging information.
* **LogManager:** The LogManager object manages the logging framework. It is responsible for reading the initial configuration parameters from a system-wide configuration file or a configuration class. The LogManager stores in a namespace hierarchy each Logger instance created with a namespace within an application. When we try to obtain the reference named logger, the LogManager class returns the already created instance of it, or creates a new instance of the named logger, stores it in a repository for future reference, and returns the new instance to the caller application.

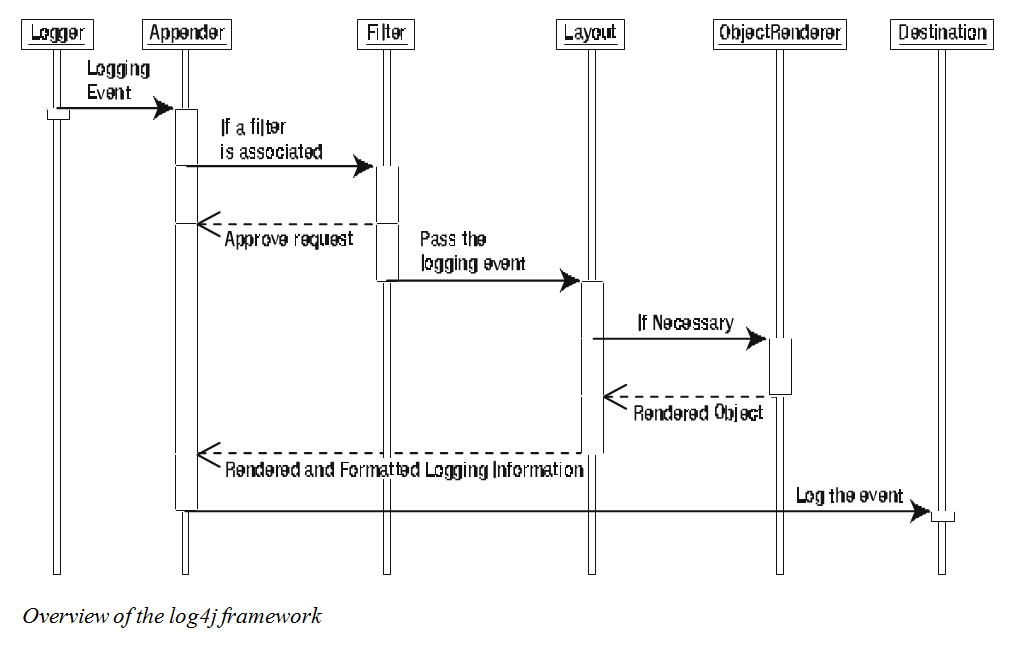
Now that you have seen the log4j core components, it is time to briefly discuss how they interact with each other.

* The central part of the log4j framework is the Logger object.
* An application instantiates a named Logger instance and passes various logging information to it.
* A Logger object has a designated Level object associated with it. The Logger object provides several logging methods

that are capable of logging information into categorized levels.

* A Logger logs only the messages with Level objects equal to or greater than its assigned Level object, or else it rejects the logging request.
* Once the Level condition has been met, the Logger object passes the logging information to all its associated Appender objects and to all the Appender objects associated with its parent Logger, recursively up the logging hierarchy.
* Similar to Logger objects, Appender objects can also have threshold Levels attached to them.The logging information is validated against the threshold Level attached to the Appender.If the log message has a Level equal to or greater than the threshold Level, the logging message is passed to the next stage.
* The Appender objects then look for any Filter object associated with them. If there are any, the logging information is passed through all the Filter objects in a chain.
* Once all the Filter objects approve a message, the Appender utilizes any Layout object associated with it to format the message, and finally it publishes the logging information to the preferred destination.

The following figure depicts the overall flow of the log4j logging architecture in a UML sequence diagram



**Naming and Placing the Configuration File**

By default, the LogManager class will look for a file named log4j.properties or log4j.xml in the classpath used for loading the log4j classes.

**Note:**Up to version 1.2.6, log4j would look for only the log4j.properties file in the classpath. Since version 1.2.7, log4j looks for both log4j.properties and log4j.xml in the classpath.

If you decide to name your configuration file differently, you are free to do that. However,you have to let the log4j framework know by supplying a command line argument as follows: **-Dlog4j.configuration="file\_name"**

**Sample Configurations**

Let’s look at a sample configuration file. The below listinng shows a simple log4j configuration.

It defines the level and appender for the root logger. You can name this file log4j.properties

and place it in the application’s classpath. The log4j framework will pick up any log4j.

properties file in the classpath.

*A Simple log4j Configuration File*

#set the level of the root logger to DEBUG and set its appender

#as an appender named testAppender

log4j.rootLogger = DEBUG, testAppender

#define a named logger

log4j.logger.dataAccessLogger = com.it.logging.logger

#set the appender named testAppender to be a console appender

log4j.appender.testAppender=org.apache.log4j.ConsoleAppender

#set the layout for the appender testAppender

log4j.appender.testAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.testAppender.layout.conversionPattern=%m%n

The preceding configuration defines the level of the root logger as DEBUG and specifies the appender to use as testAppender. Next, we define the appender testAppender as referencing the org.apache.log4j.ConsoleAppender object and specify the layout of the appender as org.apache.log4j.PatternLayout. A PatternLayout also requires that a conversion pattern or a format be supplied. We supply the conversion pattern %m%n in this case, which means the

logging message will be printed followed by a newline character.

**Note** Within the conversion pattern, %m represents the message string and %n represents a newline character.

A more complex configuration can attach multiple appenders to a particular logger. Each

appender, in turn, can have a different layout, and that layout can have a conversion pattern

associated with it. The below listing is an example of a more complex configuration file.

*Complex log4j Configuration File*

# define the root logger with two appenders writing to console and file

log4j.rootLogger = DEBUG, CONSOLE, FILE

#define your own logger named com.foo

#and assign level and appender to your own logger

log4j.logger.com.foo=DEBUG,FILE

#define the appender named FILE

log4j.appender.FILE=org.apache.log4j.FileAppender

log4j.appender.FILE.File=${user.home}/log.out

#define the appender named CONSOLE log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender

log4j.appender.CONSOLE.conversionPattern=%m%n

This configuration file defines the root logger as having level DEBUG and attaches two appenders named CONSOLE and FILE to it. We define one of our own custom loggers with the name com.foo and the level DEBUG and attach an appender named FILE to the custom logger. The appender FILE is defined as org.apache.log4j.FileAppender. The FILE appender writes to a file named log.out located in the user.home system path.

It is important to note that log4j supports UNIX-style variable substitution such as ${variableName}. The variable name defined is considered as the key and searched first in the system properties. If the log4j framework does not find the name, it then looks for the value for the variable in the properties file being parsed. The CONSOLE appender is then assigned to the org.apache.log4j.ConsoleAppender and the conversion pattern defined is %m%n, which means the printed logging message will be followed by a newline character.

**XML-Style Configuration**

With log4j, it is possible to define configuration parameters in an XML file and pass that file to the application at startup to configure different logging components. The XML configuration follows a document type definition (DTD) named log4j.dtd.

The configuration parameters and values are described in tag formats. For example, the config-

uration information in below lisitng can be defined in XML format as follows:

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

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration>

<appender name="dataAccessLogger" class="org.apache.log4j.ConsoleAppender">

<layout class="org.apache.log4j.PatternLayout">

<param name="conversionPattern" value="%m%n"/>

</layout>

</appender>

<logger name="com.it.logging.log4j" additivity="false">

<level value="debug"/>

<appender-ref ref="dataAccessLogger"/>

</logger>

</log4j:configuration>

Both the XML-style configuration and properties-style configuration are quite effective in configuring log4j. Both are flexible and good configuration styles. However, a few components in log4j, such as Filter, AsyncAppender, etc., can be configured only through XML-style configuration. It seems that log4j is evolving from properties-style configuration toward XML-style configuration, and gradually it will embrace XML-style configuration as the standard. For the time being, you need to master both configuration styles.

**Log4j Initialization**

In general, log4j makes no assumption about the environment it is running in. Therefore, it is the application developer’s responsibility to configure log4j. Configuring log4j essentially means specifying loggers, appenders, layouts, etc. The LogManager class performs the initialization operation at startup only once through a class-level static initializer block. The default initialization operation consists of the following steps:

**1.** The LogManager class looks for the system property log4j.configuration.

**2.** If the log4j.configuration property is not defined, then it tries to look for a resource named log4j.properties/log4j.xml in the application classpath.

**3.** It then attempts to convert the defined configuration resource string to a URL object. If the resource string cannot be converted into a valid java.net.URL object, then it throws a java.net.MalformedURLException.

**4.** If no resource can be found, the initialization is aborted.

**5.** If the resource is found and if it is a normal Java properties-style file containing configuration information in a key-value format, then the org.apache.log4j.PropertyConfigurator class is used to parse the configuration file. If the configuration file is an XML file, the org.apache.log4j.xml.DOMConfigurator class is used to parse the XML file and initialize the logging objects.

It is possible to avoid the default initialization classes such as LogManager and write your own configuration class. The custom configuration class can be specified as a log4j.configurationClass to the Java runtime at application startup. Any custom configuration class should implement the org.apache.log4j.spi.Configurator interface by default.