# Java Logger

In Java, **logging** is an important feature that helps developers to trace out the errors. Java is the programming language that comes with the **logging** approach. It provides a **Logging API** that was introduced in Java 1.4 version. It provides the ability to capture the log file. In this section, we are going to deep dive into the **Java Logger API**. Also, we will cover **logging level, components, Logging handlers or appenders, logging formatters or layouts, Java Logger class,**

## Components of Logging

The Java Logging components used by the developers to create logs and passes these logs to the corresponding destination in the proper format. There are the following three core components of the Java logging API:

* Loggers
* Logging Handlers or Appender
* Logging Formatters or Layouts

-------------

***Logging Frameworks***

Log4j: Apache Log4j is an open-source Java-based logging utility.

SLF4J: It stands for Simple Logging Facade for Java (SLF4J). It is an abstraction layer for multiple logging frameworks such as Log4j, Logback, and java.util.logging.

Logback: It is an open-source project designed as a successor to Log4j version 1 before Log4j version 2 was released.

tinylog(tinylog): It is a light weighted and open-source logger.

----------------

Src: <https://www.baeldung.com/java-logging-intro#Log4j>

|  |  |
| --- | --- |
| **Log Level** | **Description** |
| ALL | This level turns on all levels of logging. It includes the custom logging levels that you have defined. Once this one is configured and the levels are not considered at all, then all the appenders will start pouring the log events in log files. |
| DEBUG | Debug is used a lot for debugging the application at development time. Every log message will appear to log files once this level is set. It basically belongs to developers. |
| INFO | The INFO logging level is used to record messages about routine application operation. In real-time, system administrators watch the info logs to ensure what's happening on the system right now, and if there is any problem in normal flow. |
| WARN | WARN log level is used to indicate that you might have a problem and that you've detected an unusual situation. Maybe you were demanding to invoke a service, and it failed a couple of times before connecting on an automatic retry. It is unexpected and unusual, but no real harm was done, and it's not known whether the issue will persist or recur. Someone should investigate warnings. |
| ERROR | The ERROR log level is used to denote a serious problem that you must have to investigate immediately. Not as serious as FATAL, but still a problem. It simply means that your application has met really undesired state. For example, unexpected formatted input, database unavailability. |
| FATAL | The FATAL log level, like ERROR, designates a problem. But unlike ERROR, it designates a very serious error event. You will not consider their presence very much on a normal day, but once they appear, it signals very bad news, even the application of death. |
| OFF | This is the highest possible rank and is intended to turn off logging. |
| TRACE | This has been recently introduced in version 1.2 and includes more information to debug level logs. |

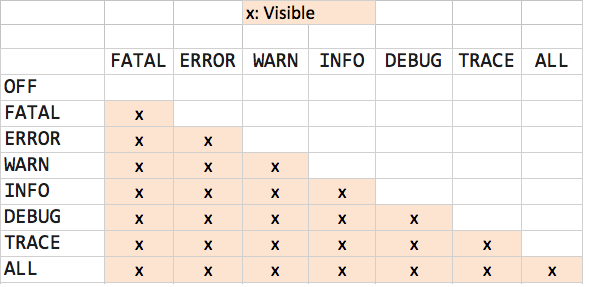
## How do Log Levels Works?

The working of logging levels is actually very simple. During runtime, the application code will create logging requests, which will have a level. At the same time, the logging framework has a log level configured, which acts as a threshold. If the request level is at the configured level or higher level, it gets logged to the configured target. If not, it's denied. It's simple as that.

Let's consider that as the following rank order for the levels:

1. ALL **<** **TRACE** **<** **DEBUG** **<** **INFO** **<** **WARN** **<** **ERROR** **<** **FATAL** **<** **OFF**

So if, for instance, the logging framework level is set to WARN, requests with any of the levels WARN, FATAL, and ERROR will be accepted, while the rest will be denied.



In the above diagram, the vertical header displays the Level of the LogEvent, while the horizontal header shows the Level associated with the appropriate logging configuration.

For the first column, you will see how the log works in each level. For example, for WARN, (FATAL, ERROR, and WARN) will be visible. For OFF, nothing will be visible.

## **1. Overview**

Logging is a powerful aid for understanding and debugging program's run-time behavior. Logs capture and persist the important data and make it available for analysis at any point in time.

This article discusses the most popular java logging framewloorks, Log4j 2 and Logback, along with their predecessor Log4j, and briefly touches upon SLF4J, a logging facade that provides a common interface for different logging frameworks.

## **2. Enabling Logging**

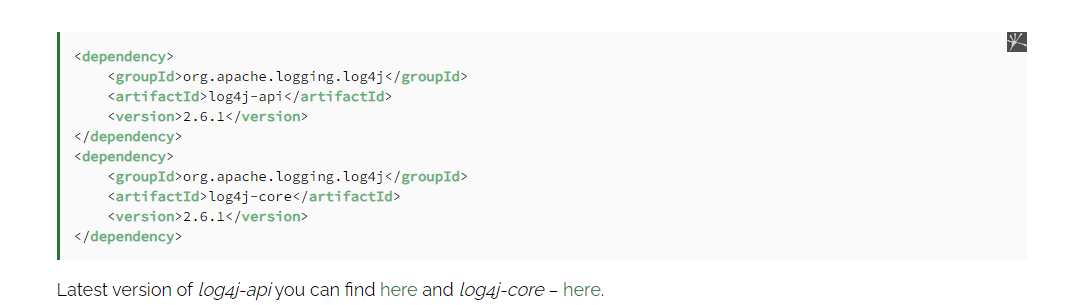
All the logging frameworks discussed in the article share the notion of loggers, appenders and layouts. Enabling logging inside the project follows three common steps:

* Adding needed libraries
* Configuration
* Placing log statements

The upcoming sections discuss the steps for each framework individually.

## **3. Log4j 2**

Log4j 2 is new and improved version of the Log4j logging framework. The most compelling improvement is the possibility of asynchronous logging. Log4j 2 requires the following libraries:



### **3.1. Configuration**

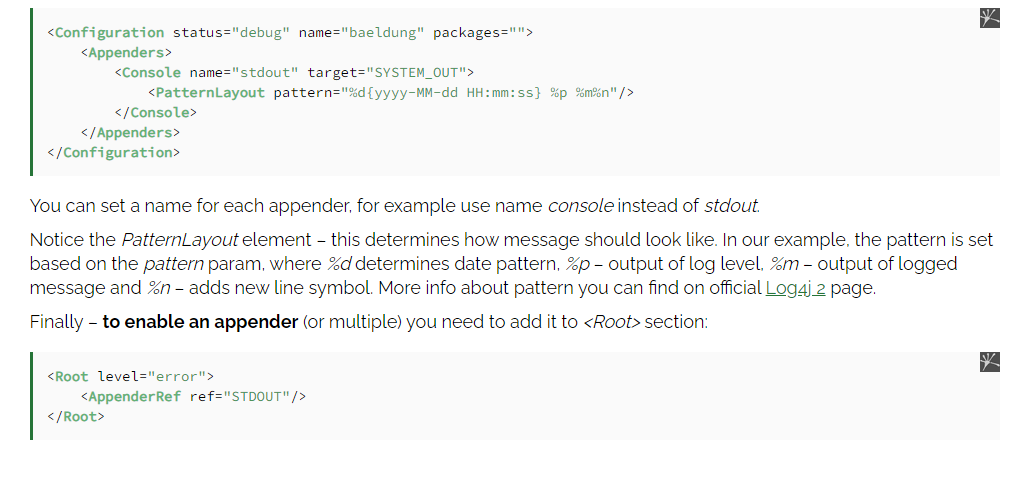
Configuring Log4j 2 is based on the main configuration *log4j2.xml* file. The first thing to configure is the appender.

These determine where the log message will be routed. Destination can be a console, a file, socket, etc.

Log4j 2 has many appenders for different purposes, you can find more information on the official [Log4j 2](https://logging.apache.org/log4j/2.x/manual/appenders.html) site.

Lets take a look at a simple config example

<https://logging.apache.org/log4j/2.x/manual/layouts.html>



### **3.2. Logging to File**

Sometimes you will need to use logging to a file, so we will add *fout* logger to our configuration:

<**Appenders**>  
 <**File** name="fout" fileName="baeldung.log" append="true">  
 <**PatternLayout**>  
 <**Pattern**>%d{yyyy-MM-dd HH:mm:ss} %-5p %m%nw</**Pattern**>  
 </**PatternLayout**>  
 </**File**>  
</**Appenders**>

The *File* appender have several parameters that can be configured:

* *file* – determines file name of the log file
* *append* – The default value for this param is true, meaning that by default a *File* appender will append to an existing file and not truncate it.
* *PatternLayout* that was described in previous example.

In order to enable *File* appender you need to add it to *<Root>* section:

<**Root** level="INFO">  
 <**AppenderRef** ref="stdout" />  
 <**AppenderRef** ref="fout"/>  
</**Root**>

### **3.3. Asynchronous Logging**

If you want to make your Log4j 2 asynchronous you need to add LMAX disruptor library to your *pom.xml*. LMAX disruptor is a lock-free inter-thread communication library.

Adding disruptor to pom.xml:

<**dependency**>  
 <**groupId**>com.lmax</**groupId**>  
 <**artifactId**>disruptor</**artifactId**>  
 <**version**>3.3.4</**version**>  
</**dependency**>

Latest version of disruptor can be found [here](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22com.lmax%22%20AND%20a%3A%22disruptor%22).

If you want to use LMAX disruptor you need to use *<asyncRoot>* instead of *<Root>* in your configuration.

<**AsyncRoot** level="DEBUG">  
 <**AppenderRef** ref="stdout" />  
 <**AppenderRef** ref="fout"/>  
</**AsyncRoot**>

Or you can enable asynchronous logging by setting the system property *Log4jContextSelector* to *org.apache.logging.log4j.core.async.AsyncLoggerContextSelector*.

You can of course read more about the configuration of the Log4j2 async logger and see some performance diagrams on the [Log4j2 official page](https://logging.apache.org/log4j/2.x/manual/async.html) .

### **3.4. Usage**

The following is a simple example that demonstrates the use of Log4j for logging:

**import** org.apache.logging.log4j.Logger;  
import org.apache.logging.log4j.LogManager;  
  
public **class** **Log4jExample** {  
  
 **private** **static** **Logger** logger = LogManager.getLogger(Log4jExample.class);  
  
 **public** **static** **void** **main**(String[] args) {  
 logger.debug("Debug log message");  
 logger.info("Info log message");  
 logger.error("Error log message");  
 }  
}

After running, the application will log the following messages to both console and file named *baeldung.log:*

2016-06-16 17:02:13 INFO Info log message  
2016-06-16 17:02:13 ERROR Error log message

If you elevate the root log level to *ERROR*:

<**level** value="ERROR" />

The output will look like the following:

2016-06-16 17:02:13 ERROR Error log message

As you can see, changing the log level to upper parameter causes the messages with lower log levels will not be print to appenders.

Method *logger.error* can be also used to log an exception that occurred:

**try** {  
 // Here some exception can be thrown  
} **catch** (Exception e) {  
 logger.error("Error log message", throwable);  
}

### **3.5. Package Level Configuration**

Let's say you need to show messages with the log level TRACE – for example from a specific package such as *com.baeldung.log4j2*:

logger.trace("Trace log message");

For all other packages you want to continue logging only INFO messages.

Keep in mind that TRACE is lower than the root log level INFO that we specified in configuration.

To enable logging only for one of packages you need to add the following section before *<Root>* to your *log4j2.xml*:

<**Logger** name="com.baeldung.log4j2" level="debug">  
 <**AppenderRef** ref="stdout"/>  
</**Logger**>

It will enable logging for *com.baeldung.log4j* package and your output will look like:

2016-06-16 17:02:13 TRACE Trace log message  
2016-06-16 17:02:13 DEBUG Debug log message  
2016-06-16 17:02:13 INFO Info log message  
2016-06-16 17:02:13 ERROR Error log message