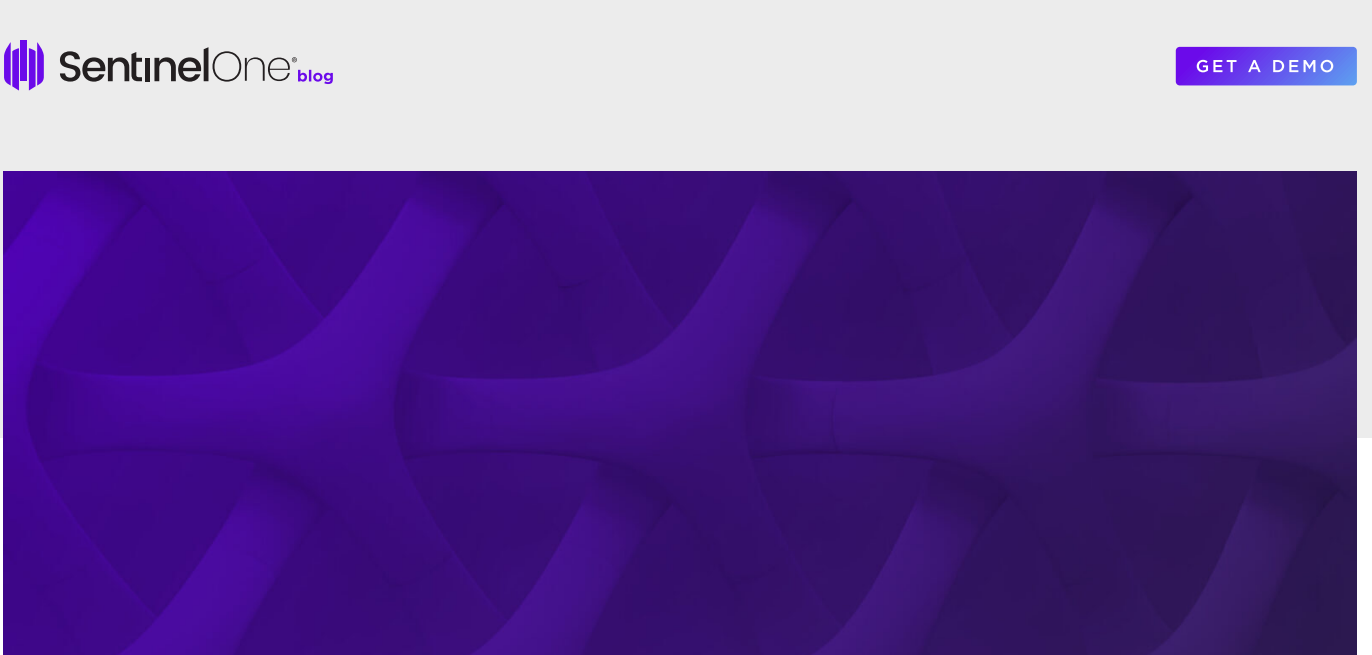


Log4J2 Configuration: A Detailed Guide to Getting Started

September 5, 2018
by SentinelOne

One of the most annoying aspects of software development is definitely logging. If a non-trivial application lacks logging, then whoever is maintaining it will struggle and the post-mortem debug will be mostly a guessing game. Today we present another contribution to remedying this situation by offering a tutorial on log4j2 configuration.

There are many ways to log in Java. You can use a more manual approach, but the recommended route is the adoption of a dedicated logging framework. That's why we're covering Apache log4j2 which, being an improved version of log4j, is one of the industry-standard logging frameworks.

We'll start by making a quick recap of our previous post on Java logging and introducing some facts about log4j2. Then we'll proceed to cover the state of the sample application we've started writing in the previous tutorial.

After that, we get to the meat of the article. You'll learn how to configure log4j2, starting with the basics, and progressing through more advanced topics, such as log formatting, appenders, log levels, and log hierarchies.

Let's get started.

Logging With Log4j2: Not Our First Rodeo

We covered basic logging for Java applications a while back. In that tutorial, we used log4j version 2, a logging framework from the Apache project.

Let's go one step further with Java application logging and look at log4j2 configuration. Today we'll cover the basic aspects of log4j2 configuration to help you get started.

Log4j's capabilities have made it one of Java's most popular logging frameworks. It can be configured for multiple logging destinations and a variety of log file formats.

Log4j's capabilities have made it one of Java's most popular logging frameworks.

Log messages can be filtered and directed at the individual class level, giving developers and operations personnel granular control over application messages.

Let's examine these mechanisms by configuring log4j with a command line Java application.

Sample Application

Let's start where we left off in the previous tutorial, with an application that logs with log4j.

```
package com.company;

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

public class Main {

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

  public static void main(String[] args) {
      String message = "Hello there!";
      logger.trace(message);
      logger.debug(message);
      logger.info(message);
      logger.warn(message);
      logger.error(message);
      logger.fatal(message);
  }
}
```

This is similar to the application at the end of the previous post, with a few additional logging statements. We're logging the same message at each of log4j's predefined logging levels: trace, debug, info, warn, error, and fatal.

We will be using log4j's YAML file format, so you'll need to add a few additional dependencies to your **pom.xml** (or **build.gradle**).

```
<dependencies>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.12.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.12.1</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-yaml</artifactId>
        <version>2.10.0</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.10.0</version>
    </dependency>
</dependencies>
```

Set this code up so you can build and run it using your favorite Java tools.

Essential Log4j2 Configuration

Default Configuration

Let's run our application **without a log4j configuration file.** If you already have one, delete it or move it to another file name so that log4j will ignore it.

When we run the application, we see this on the console:

```
09:38:14.114 [main] ERROR com.company.Main - Hello there!
09:38:14.119 [main] FATAL com.company.Main - Hello there!
```

Two of the six log messages, the ones specified as "error" and "fatal," are sent to the console.

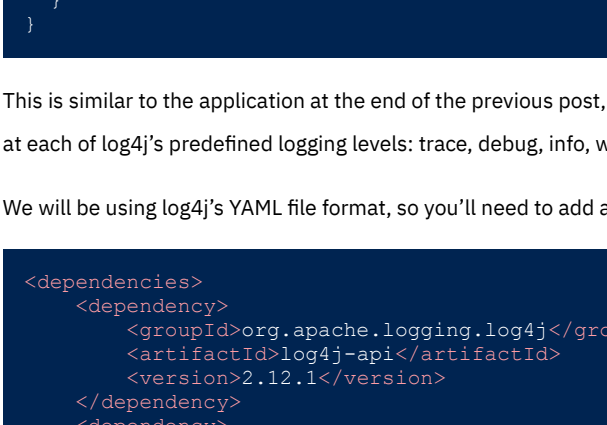
Log4j has a default configuration. It will log to the console, showing messages classified as "error" or higher.

Knowing how log4j will behave without a configuration file is useful, but let's look at how to set it up for our needs.

Configuration File Location

We can provide log4j with a configuration file in a specific location via the **log4j.configurationFile** system property. This is the first place it will look for a configuration file.

If log4j cannot find the system property, it looks for a file in the classpath. Since log4j version 2 supports four different file formats and two different file naming conventions, the rules for locating a file are complicated. We'll go over them after we cover the different options.



Configuration File Formats

Log4j will load Java properties and YAML, JSON, and XML configuration files. It identifies the file format by examining the file extension.

1. Java properties — **.properties**
2. YAML — **.yaml** or **.yml**
3. JSON — **.json** or **.jsn**
4. XML — **.xml**

A file specified by the **log4j.configurationFile** system property must have one of these file extensions but can have any base name. Log4j will parse it based on the format indicated by the extension.

When log4j scans the classpath for a file, it scans for each format in the order listed above and stops when it finds a match.

For example, if it finds a YAML configuration, it will stop searching and load it. If there is no YAML file but it finds JSON, it will stop searching and use it instead.

Configuration File Names

When log4j scans the classpath, it looks for one of two filenames: log4j2-test.[extension] or log4j2.[extension].

It loads test files first, giving developers a convenient mechanism for forcing an application to log at debug or trace level without altering the standard configuration.

Scanning for Configuration

When we put the rules for file formats and names together, we can see log4j's algorithm for configuring itself.

If any of the following steps succeed, log4j will stop and load the resulting configuration file.

1. Check for the **log4j.configurationFile** system property and load the specified file if found.
2. Search for log4j2-test.properties in the classpath.
3. Scan classpath for log4j2-test.yaml or log4j2-test.yml
4. Check for log4j2-test.json or log4j2-test.jsn
5. Search for log4j2-test.xml
6. Look for log4j2.properties
7. Search for log4j2.yaml or log4j2.yml
8. Scan classpath for log4j2.json or log4j2.jsn
9. Check for log4j2.xml
10. Use the default configuration.

Practice Proper Configuration File Hygiene

There are 12 potential configuration file names for log4j. Loading the wrong one can lead to lost logging information or diminished performance if an application logs unnecessary messages in a production environment.

Before deploying code, make sure your application has one and only one configuration file and that you know where it is. If you insist on loading configuration from the classpath, scan for spurious files before releasing your code.

Basic Configuration

Now that we know how to supply a configuration to log4j, let's create one and use it to customize our application.

Log4j's Default Configuration Revisited

Let's start with the default configuration and modify our application's behavior from there. We'll take the hint from log4j's configuration rules and use YAML.

The default configuration looks like this:

```
Configuration:
  status: warn
  Appenders:
    Console:
      name: Console
      target: SYSTEM_OUT
      PatternLayout:
        Pattern: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
  Loggers:
    Root:
      level: error
      AppenderRef:
        ref: Console
```

Create a file name **log4j2.yaml** with these contents and set **log4j.configurationFile** to point to its location.

Next, run the application. You'll see the same output as before.

```
09:38:14.114 [main] ERROR com.company.Main - Hello there!
09:38:14.119 [main] FATAL com.company.Main - Hello there!
```

We've taken control of our application's logging configuration. Now let's improve it.

Log File Location

The first step is to get our logs off of the console and into a file. To do this, we need to understand **appenders.**

Appenders put log messages where they belong. The default configuration supplies a **console appender.** As the name suggests, it appends messages to the console.

```
Appenders:
  Console:
    name: Console
    target: SYSTEM_OUT
    PatternLayout:
      Pattern: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
```

We want a **file appender.** Let's replace our console appender.

```
Appenders:
  File:
    name: File_Appender
    fileName: logfile.log
    PatternLayout:
      Pattern: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
```

File appenders have a **name**, just like console appenders. But instead of a **target**, they have a **fileName**.

Similar to console appenders, they also have a **PatternLayout,** which we will cover below.

The name is not just for show. If we want to replace the console appender with the file appender, we need to let our **logger** know where to put our log messages.

So change the **ref** value in the logger to the file appender's name.

```
Loggers:
  Root:
    level: error
    AppenderRef:
      ref: File_Appender
```

Now, rerun the application. Instead of logging to the console, it places the messages in a file named **logfile.log** in the working directory. We've moved our logs to a file!

Logging Levels

We still only saw two of our six log messages in our log file. Let's talk about **loggers** and how they manage log messages.

Our basic configuration defines a single logger.

```
Loggers:
  Root:
    level: error
    AppenderRef:
      ref: File_Appender
```

It has a level of "error," so it only prints messages that are errors or fatal.

When a logger receives a log message, it passes it on or filters it based on its configured level. This table shows the relationship between logger configuration and log message level.

So if we change the level of our logger, we'll see more messages. Set it to "debug."

```
Loggers:
  Root:
    level: debug
    AppenderRef:
      ref: File_Appender
```

Next, rerun the program. The application logs all of the messages that are debug level or higher.

Logger Hierarchy

Log4j arranges loggers in a hierarchy. This makes specifying a different configuration for individual classes possible.

Let's change our application and see this in action.

```
public class Main {

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

  public static void main(String[] args) {
    String message = "Hello there!";
    System.out.println(message);
    logger.debug(message);
    logger.info(message);
    logger.error(message);

    LoggerChild.log();
  }

  private static class LoggerChild {
    private static final Logger childLogger = LogManager.getLogger(LoggerChild.class);

    static void log() {
        childLogger.debug("Hi Mom!");
    }
  }
}
```

We've added an inner class that creates a logger and logs a message with it.

After **Main** does its logging, it calls **LoggerChild.**

If we run this with our current config, we see the new message and that it's logged from a different class.

```
12:29:23.325 [main] DEBUG com.company.Main - Hello there!
12:29:23.331 [main] INFO  com.company.Main - Hello there!
12:29:23.332 [main] ERROR com.company.Main - Hello there!
12:29:23.333 [main] DEBUG com.company.Main.LoggerChild - Hi Mom!
```

Loggers have a class hierarchy similar to Java's. All loggers are descendants of the **Root** logger we've been working with so far.

Loggers that lack any specific configuration inherit the **Root** configuration.

So when **Main** and **LoggerChild** create loggers using their class name, these loggers inherit **Root's** configuration, which is to send debug level and higher messages to the **File_Appender.**

We can override this specifying configuration for the two loggers.

```
Loggers:
  logger:
    -
      name: com.company.Main
      level: error
      additivity: false
      AppenderRef:
        ref: File_Appender
    -
      name: com.company.Main.LoggerChild
      level: debug
      additivity: false
      AppenderRef:
        ref: File_Appender
  Root:
    level: debug
    AppenderRef:
      ref: File_Appender
```

Loggers are named in the **logger** section. Since we're listing two, we use the YAML array syntax.

We set **com.company.Main's** logger to "error" and **com.company.Main.LoggerChild's** to "debug."

The **additivity** setting controls whether or not log4j will send messages from a logger's ancestor to a descendant.

If set to true, both loggers will process the same message. Some systems want to add the same message to two different logs. We don't want this behavior, so we've overridden the default and specified **false.**

Now run the program again:

```
12:33:11.062 [main] ERROR com.company.Main - Hello there!
12:33:11.073 [main] DEBUG com.company.Main.LoggerChild - Hi Mom!
```

We only saw the error message from **Main** but still saw the debug message from **LoggerChild**!

More Than One Appender

Just like we can have more than one logger, we can have more than one appender.

Let's make a few changes to our configuration.

Add a second file appender. To do this, create a list with the original appender and the second one with a different name and file. Your **Appenders** section should look like this:

```
  Appenders:
    File:
      -
        name: File_Appender
        fileName: logfile.log
        PatternLayout:
          Pattern: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
      -
        name: Child_Appender
        fileName: childlogfile.log
        PatternLayout:
          Pattern: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
```

Next, point the LoggerChild logger at the new appender. Your **Loggers** section will look like this.

```
 Loggers:
   logger:
     -
      name: com.company.Main
      level: error
      additivity: false
      AppenderRef:
        ref: File_Appender
     -
      name: com.company.Main.LoggerChild
      level: debug
      additivity: false
      AppenderRef:
        ref: Child_Appender
   Root:
     level: debug
     AppenderRef:
      ref: File_Appender
```

Now run the application and you'll see two different log files, each with the messages from their associated classes.

Log Message Formatting

Each of our appenders has a PatternLayout.

```
PatternLayout:
  Pattern: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
```

PatternLayout is an instance of a Log4j layout class. Log4j has built-in layouts for logging messages in CSV, JSON, Syslog, and a variety of different formats.

PatternLayout has a set of operators for formatting messages that operates similarly to C's **sprintf** function. By specifying a pattern, we control the format of log messages when they are written by the appender.

Our layout string looks like this:

```
"%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
```

Each % corresponds to a field in a log message.

Format Specifier	Description
%d{HH:mm:ss.SSS}	date as hour :minute "seconds . milliseconds
%t	thread name
%–5level	log level, right-padded to 5 spaces
%logger{36}	logger name, up to 36 package levels deep
%msg	log message
%n	carriage return

There are many additional operators for PatternLayout.

Variable Replacement

Configuration files can become repetitive as appenders and loggers multiply. Log4j supports variable substitution to help reduce repetition and make them easier to maintain. Let's refine our configuration with the use of **Properties**.

```
Configuration:
  status: warn
  Properties:
    property:
      -
        name: "LogDir"
        value: "logs"
      -
        name: "DefaultPattern"
        value: "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"
  Appenders:
    File:
      -
        name: File_Appender
        fileName: ${LogDir}/logfile.log
        PatternLayout:
          Pattern: ${DefaultPattern}
      -
        name: Child_Appender
        fileName: ${LogDir}/childlogfile.log
        PatternLayout:
          Pattern: ${DefaultPattern}
  Loggers:
    logger:
      -
        name: com.company.Main
        level: error
        additivity: false
        AppenderRef:
          ref: File_Appender
     -
        name: com.company.Main.LoggerChild
        level: debug
        additivity: false
        AppenderRef:
          ref: Child_Appender
    Root:
      level: debug
      AppenderRef:
        ref: File_Appender
```

At the top of the file, we declared two **Properties,** one named **LogDir** and another **DefaultPattern**.

After declaring a property, it can be used in the configuration using braces and a dollar sign: **${LogDir}** or **${DefaultPattern}**

LogDir is a subdirectory name we added to the names of the two log files. When we run the application, log4j will create this directory and place the log files there.

We specified **DefaultPattern** as the pattern layout for our two log files, moving the definition to one place. If we want to modify our log file format, we only have to worry about changing it once now.

Log4j can also import properties from the environment. You can find the details here.

For example, if we want to import the log file directory from a Java system property we specify it as **${sys:LogDir}** in the log4j configuration and set a LogDir system property to the desired directory.

Automatic Reconfiguration

Log4j can reload its configuration at a periodic interval, giving us the ability to change an application's logging configuration without restarting it.

Add the **monitorInterval** setting to the **Configuration** section of the file and log4j will scan the file at the specified interval.

```
Configuration:
  monitorInterval: 30
```

The interval is specified in seconds.

Conclusion

Log4j is a powerful logging framework that allows us to configure our applications to log in a variety of different ways, with granular control over how different components use log files.

This tutorial covered the basic aspects of configuring log4j, but there's much more to learn. You can learn about log4j configuration on the project's website.

Log4j is a powerful logging framework.

To learn more about Java logging and logging strategies in general, you're already in the right place! Scalyr's blog has many more tutorials and reference guides like this one.

Scalyr offers a log aggregation tool, which means that once you have lots of log files and data, they'll help you organize, search, and make sense of all these data. So stay tuned for more!

This post was written by Eric Goebelbecker. Eric has worked in the financial markets in New York City for 25 years, developing infrastructure for market data and financial information exchange (FIX) protocol networks. He loves to talk about what makes teams effective (or not so effective!)

Like this article? Follow us on LinkedIn, Twitter, YouTube or Facebook to see the content we post.

Read more about Cyber Security

- Scalyr updates: Tabular search results, log lines in alerts and more
- Scalyr and the Log4j Vulnerability