# Integrating 3rd Party Java Logging Frameworks into SAP's Logging Framework

## Applies to:

SAP NetWeaver Composition Environment 7.2

## Summary

Logging is a means for applications to inform system administrators or developers about their current state or state changes. Java application developers typically use the Log4j or Apache Commons Logging (JCL) frameworks to log information. Both logging frameworks are open source, platform independent and widely used in many projects. SAP's business solutions use a proprietary SAP logging API, which is integrated into the SAP Solution Manager. This ensures that SAP support teams can access log information at customer sites in urgent cases. It is recommended that Java applications running on SAP NetWeaver use the Logging API, since all logs are then accessible in one common format within one log viewer.

When developing Java applications on SAP NetWeaver developers can choose between any of these three logging frameworks among others of course. There are good reasons for each framework. This tutorial discusses the following three use cases:

- Logging with Log4j / Apache Commons Logging (JCL)
- Logging with SAP's Logging API
- Integrating Log4j / Apache Commons Logging into SAP's Logging API

This tutorial shows how the Java logging frameworks can be configured to log into SAP's logging infrastructure without instrumenting any existing source code. The implementation of a required bridge that routes Log4j/JCL logs to SAP's logging infrastructure will be shown. Hence integrating 3rd party logging products into SAP's logging framework can be done with an absolutely feasible effort for almost every Java development scenario.

**Author(s):** Peter Kulka

**Company:** SAP AG

**Created on:** October 5, 2010

## Author Bio

Peter Kulka received his PhD in Computer Science / Computer Graphics from the University of Auckland, New Zealand. After teaching Java / C++ programming courses at the University of Auckland, Peter joined SAP AG in 1999. Until November 2005 Peter was responsible for the product definition of the SAP NetWeaver Application Server, SAP's Java EE 5 compliant application server. Currently Peter is a Solution Architect in SAP's Global Ecosystem and Partner Group, where he advises SAP partners on how to architect their Java solutions based on SAP NetWeaver. Peter presented at leading industry conferences such as JavaOne, SAP TechEd, and OOP.

## Table of Contents

## Introduction

Administrators, developers and support organizations view log entries to get information about the health of their applications and systems. In the Java community Log4j or Apache Commons Logging (JCL) are probably the most popular and widely used logging frameworks.

The Log4j logging framework provides for developers an abstraction from logging details, such as the format of log entries or the location of log files. The framework consists of Loggers, Layouts (or Formatters / Renderers) and Appenders (or Handlers). The Loggers are logical log file names used by the Java source code to identify the different loggings. Each Logger may be assigned to a Layout, which formats the log entries. The formatted log entries are passed to Appenders, which are Java classes that write the actual log files. Examples for Appenders that come with Log4j are the `FileAppender`, `ConsoleAppender`, `SocketAppender` and `SMTPAppender`. The assignment of Loggers to Layouts and Appenders is done through configuration (configuration file: `log4j.properties`). This means that the Java developer, who uses the Log4j API, does not need to worry about the logging details, since this can later be configured. Log4j is used by many open source projects, such as JBoss or TheServerSide.

While Log4j provides an abstraction to the logging details, Apache Commons Logging (JCL) goes one step further an abstracts the Java code from the underlying logging framework, such as Log4j. So using JCL the logging framework can be switched by configuration (in the `commons-logging.properties` file) without changing the source code. This is particular useful for Java frameworks such as the Apache Struts Web application framework. The combination of JCL and Log4j is very popular.

SAP's logging API distinguishes between logging and tracing. Log files (with the extension `.log`) are intended for administrators, while trace files (with the extension `.trc`) are supposed to be viewed by developers and support people. Log messages are subdivided into categories, which must be provided when calling the logging API. The top-level categories are `Archive`, `Services`, `System` and `Applications`. Using categories helps administrators to quickly find the relevant log entries. Since trace messages are meant to be viewed by developers and support people, they contain very detailed information. To quickly find the relevant trace entry, trace messages are organized by code packages. The default location of the log and trace files is

*<SAP_install_directory>*`/`*<system_name>*`/`*<instance_name>*`/j2ee/cluster/server`*<number>*`/log/`.

In the following discussion we will focus on Log4j and JCL as examples for Java logging frameworks. The discussions and code examples presented in this tutorial should also be useful for many other logging frameworks.

### Typical Logging Use Cases

When developing Java application on SAP NetWeaver developer can choose to use Log4j, JCL or SAP's logging API. The different use cases are discussed in the following paragraphs.

#### Logging with Log4j / Apache Commons Logging (JCL)

Using Log4j or JCL on SAP NetWeaver does not require any SAP-specific adjustments. Simply place the configuration files (i.e. the `log4j.properties` or `commons-logging.properties` files respectively) together with the corresponding libraries (`log4j-`*<version>*`.jar` or `commons-logging-`*<version>*`.jar`) in the build class path of your application. That is it! The examples in this tutorial use Log4j version 1.2.16 and JCL version 1.1, which are the latest versions available at the time this tutorial was written.

Developers, who are either migrating existing Java applications to SAP NetWeaver or developing Java applications for multiple platforms, will certainly do not want to use proprietary logging frameworks to keep their applications platform independent. However, we will later show how to integrate these logging frameworks into SAP's logging API.

#### Logging with SAP's Logging API

All SAP business solutions use SAP's logging API to provide the log entries in one common format in one central place. The log viewer within the SAP NetWeaver Administrator can access all log and trace files centrally and provides comprehensive search and filter capabilities to find the relevant log entries quickly. It is recommended that all applications running on SAP NetWeaver use SAP's logging API. SAP's logging framework is integrated with the SAP Solution Manager, so that SAP support teams can access logs and



 

traces in urgent cases. For the certification of 3rd party products through the SAP Integration and Certification Center, the usage of SAP's logging API is mandatory.

Switching from an existing logging framework to SAP's logging API requires the instrumentation of the entire source code, i.e. every logging API call needs to be changed. This is simply impractical and unfeasible. Therefore the next paragraph discusses a more suitable solution.

### Integrating Log4j / Apache Commons Logging into SAP's Logging API

The goal of this paragraph is to show exemplary for Log4j and JCL, how 3rd party logging frameworks can be configured to use SAP's logging framework, so that no code instrumentation is necessary. The key idea is to provide a logging bridge for each logging framework that routes the log messages from the 3rd party logging framework to SAP's logging API.

#### Logging Bridge for Log4j

The first step is to import the Log4j configuration file (`log4j.properties`) and the Log4j library (`log4j-<version>.jar`) into a project within the SAP NetWeaver Development Studio and to include these files to the build class path of your application. The next step is to configure Log4j, so that our logging bridge is used as an Appender. Here is the content of the corresponding configuration file (`log4j.properties`):

```
# Configuration of: 1) rootLogger Log-Level 2) Appenders
log4j.rootLogger=all, SAPLogging

#=== Configuration SAP Logging Appender ===

# The appender class
log4j.appender.SAPLogging=com.sap.logging.bridge.log4j.SapLogAppender
# Name of the SAP Logging Category under "Applications"
log4j.appender.SAPLogging.categoryName=MyCategory
```

Example configuration file for Log4j: `log4j.properties`.

Log4j arranges the different Loggers in a hierarchy. The configuration file above tells Log4j that the top-level Logger (the `rootLogger`) should write all log messages to the Appender `SAPLogging` (line2). Line 7 defines the Appender `SAPLogging` by assigning a Java class to it (in this case the class `SapLogAppender`, our logging bridge). The last line defines the parameter `categoryName` of our logging bridge. Please note that this line does not belong to the Log4j configuration. We will use this parameter later, when we call the SAP logging API to set the category of our log messages (within the top-level log category `Application`, see above).

The property `rootLogger` determines the severity levels that are logged. The severity levels will be explained later. We recommend to set this property to "`all`" and use the configuration capabilities of the log viewer within the SAP NetWeaver Administrator to specify the severity level for all Java applications centrally.

The next step is to map the severity level of log messages from Log4j to SAP's logging framework. A severity level is an integer value indicating the importance of the log message. Log4j defines the following severity level (however, custom level can be added): `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG` and `TRACE`. SAP's logging framework defines pretty similar severity levels that are explained here (-> "How to Write Log and Trace Messages"; for Logging: -> "Logging" -> "Severities for Log Messages"; for Tracing: -> "Tracing" -> "Severities for Trace Messages"). For our example logging bridge, we use the following severity level mapping:

| Log4j | SAP logging framework |
|---|---|
| TRACE | DEBUG (written to trace file) |



 

| | |
|---|---|
| DEBUG | DEBUG (written to trace file) |
| INFO | INFO (written to log file, category from Log4j configuration file) |
| WARN | WARNING (written to log file, category from Log4j configuration file) |
| ERROR | ERROR (written to log file, category from Log4j configuration file) |
| FATAL | FATAL (written to log file, category from Log4j configuration file) |
| Custom level | WARNING (written to log file, category from Log4j configuration file) |

Example mapping of the severity level.

This means that the Log4j severity levels TRACE and DEBUG are mapped to the severity level DEBUG and the tracing API of SAP's logging framework is called. All other Log4J severity levels are mapped to the corresponding severity levels in SAP's logging framework, but the logging API is called instead. This mapping was chosen, so that all log messages belong to the Category "MyCategory", which is part of the top-level category "Application". So all log entries can easily found in the log view under the category "MyCategory".

Note that this mapping is just an example. Real world projects may use additional custom-defined severity levels, which have to be mapped to the existing levels of SAP's logging API. Furthermore, each severity level must also be mapped to either the logging or the tracing API. This requires a good understanding of the severity level semantics of both logging frameworks.

The relevant parts of Java code for the logging bridge is listed below. Most of this coding should be self-explaining. Here we just explain the most relevant parts. The logging bridge is a Log4j Appender, so that the corresponding interface (AppenderSkeleton) needs to be implemented. The only method that is called by the Log4j framework is the append() method. The methods logToSap() and mapSeverity() are just helpers to call SAP's logging and tracing API as well as to map the severity level between the to logging frameworks. Depending on the severity level, the logToSap() method calls either the tracing API (for the severity level "debug") or the logging API otherwise. The case statement in the logToSap() method performs the actual severity level mapping. Both the calls of the logging/tracing APIs and the severity level mapping may need to be adjusted in a Java development project. The property "categoryName", which is used to call SAP's logging API, is set by the Log4j configuration.

```
package com.sap.logging.bridge.log4j;

import ...;

/**
 * By extending the AppenderSkeleton you're able to define your own
 * destination for the log4j log messages. This SapLogAppender writes the
 * log4j messages to the SAP logging API. The appender tries to routes the
 * messages to the SAP logging API by using the SAP API as it meant to be.
 */
public class SapLogAppender extends AppenderSkeleton {
      /** Name for the category under
                     \System\Applications\<category_Name>" */
      private static String categoryName;

      /**
       * Writes the logging events of log4j to SAP logging API.
       *
       * @param event
       *            log4j log event written to log4j logger
       */
```



 

```
      protected void append(LoggingEvent event) {

            // Text of the log4j message
            String msg = event.getMessage().toString();

            // The Throwable of this logging event (if there's one)
            Throwable ta = event.getThrowableInformation() == null
                ? null
                : event.getThrowableInformation().getThrowable();

            // map SAP logging Severity
            int severity = mapSeverity(event.getLevel());

            // write log message
            logToSap(event.getLoggerName(), msg, ta, severity);
      }

      private void logToSap(String loggerName,
                            String msg,
                            Throwable ta,
                            int severity) {

            // Location for the SAP log messages
            Location loc = Location.getLocation(loggerName);
            Category cat = Category.getCategory(Category.APPLICATIONS,
                        getCategoryName());

            // in case of a unknown severity -> set severity WARNING
            // and add information
            if (severity == -1) {
                  severity = Severity.WARNING;
                  msg += " (Couldn't identify severity of log4j-Logging
Event!)";
            }

            // log-level DEBUG will be written to default trace
            if (severity == Severity.DEBUG) {
                  if (ta == null)
                        loc.logT(severity, msg);
                  else // for log messages with throwables
                        loc.traceThrowableT(severity, msg, ta);
            // everything else will be written to the application.log
            // file in the configured category
            } else {
                  if (ta == null)
                        cat.logT(severity, loc, msg);
                  else // for log messages with throwables
                        cat.logThrowableT(severity, loc, msg, ta);
            }
      }

      /**
       * Maps log4j's level to SAP logging severity.
       *
       * @param level-object
       *            of log4j-LoggingEvent
       * @return Mapped SAP severity; -1 for levels that couldn't be
       * mapped
       */
      private int mapSeverity(Level level) {
```



 

```
            switch (level.toInt()) {

            case Level.TRACE_INT:
                  return Severity.DEBUG;

            case Level.DEBUG_INT:
                  return Severity.DEBUG;

            case Level.INFO_INT:
                  return Severity.INFO;

            case Level.WARN_INT:
                  return Severity.WARNING;

            case Level.ERROR_INT:
                  return Severity.ERROR;

            case Level.FATAL_INT:
                  return Severity.FATAL;

            default: // unknown log level
                  return -1;
            }
      }

      ...
}
```

Source code of the logging bridge for Log4j.

The following code is a little Log4j example application to test the logging. After obtaining a logger for the class `Log4jExampleUsages` the method `logSomething()` simply calls the Log4j logging methods, i.e. `trace()`, `debug()`, `info()` and so on.

```
package org.example.app;

import org.apache.log4j.Logger;
/**
 * Does some logging with log4j API for demonstration.
 */
public class Log4jExampleUsage {

      /** The log4j-Logger*/
      private static final Logger log4j =
              Logger.getLogger(Log4jExampleUsage.class);

      /**
       * Just some log4j log output.
       */
      public static void logSomething() {

            // traces
            log4j.trace("Log4j - trace-method");
            log4j.debug("Log4j - debug-method");
            log4j.debug(
                  "Log4j - debug-method with exception",
                  new NullPointerException("Developer's best friend."));
```



 

```
            //logs
            log4j.info("Log4j - info-method");
            log4j.warn("Log4j - warn-method");
            log4j.error("Log4j - error-method");
            log4j.fatal("Log4j - fatal-method");
            log4j.fatal(
                  "Log4j - fatal-method with exception",
                  new NullPointerException("Developer's best friend."));

      }
}
```

Example application to produce some Log4j log messages.

This JSP script finally calls the Log4j test application, i.e. the method `logSomething()`:

```
...
<html>
<head>
<title>LOG4J to SAP LOGGING</title>
</head>
<body>
            <% org.example.app.Log4jExampleUsage.logSomething(); %>
            <h1>Log4j has written to SAP logging.</h1>
            <h1>Check your logs!</h1>
</body>
</html>
```

JSP that calls the example Log4j application.

After running the JSP script in a browser, the Log4j logs should be written to SAP's logging framework. This can be verified by the log viewer of the SAP NetWeaver Administrator.

### Logging Bridge for Apache Commons Logging (JCL)

The integration of JCL into SAP's logging framework is very similar to the integration of Log4j. As a first step we need to configure the class that implements the JCL logging API. This can be done by the following JCL configuration file (`commons-logging.properties`), which must be added to the build class path. It only specifies that the class `SapLogJclImpl` implements the JCL logging API and shall be used for logging.

```
org.apache.commons.logging.Log=com.sap.logging.bridge.jcl.SapLogJclImpl
```

Example configuration file for JCL: `commons-logging.properties`.

In contrast to the Log4j example, the configuration file does not contain the category for SAP's logging API. For the sake of simplicity of our example the category name is hard-coded in the logging bridge, see below. The next step is to define the severity level mapping between the two logging frameworks. Similar to the Log4j example, we use the following mapping:

| JCL method | SAP logging framework |
|---|---|
| `trace()` | DEBUG (written to trace file) |
| `debug()` | DEBUG (written to trace file) |



 

| | |
|---|---|
| `info()` | `INFO` (written to log file, category "`MyCategory`") |
| `warn()` | `WARNING` (written to log file, category "`MyCategory`") |
| `error()` | `ERROR` (written to log file, category "`MyCategory`") |
| `fatal()` | `FATAL` (written to log file, category "`MyCategory`") |

The log entries provided by the `trace()` and `debug()` JCL methods are mapped to the severity level "`debug`", while all other level are mapped to the corresponding severity level of SAP's logging framework.

The source of the `SapLogJclImpl` class is shown below. The main methods that must be implemented are the various logging methods `trace()`, `debug()`, `info()` and so on. All of them just map the severity level and call the `logToSap()` method, which calls the SAP logging API.

```
package com.sap.logging.bridge.jcl;

import ...;

public class SapLogJclImpl implements Log, Serializable {

      private static final long serialVersionUID = 1377781809226327977L;
      Location loc;
      /** Name for the category under
                    "\System\Applications\<category_Name>" */
      final String CATEGORY_NAME = "MyCategory";
      Category cat = Category.getCategory(Category.APPLICATIONS,
                                          CATEGORY_NAME);
      ...

      public SapLogJclImpl(String name) {
            loc = Location.getLocation(name);
      }

      public boolean isDebugEnabled() {
            return Severity.DEBUG >= loc.getEffectiveSeverity();
      }

      ...

      public void trace(Object msg) {
            logToSap(msg.toString(), null, Severity.DEBUG);
      }

      public void trace(Object msg, Throwable ta) {
            logToSap(msg.toString(), ta, Severity.DEBUG);
      }

      public void debug(Object msg) {
            logToSap(msg.toString(), null, Severity.DEBUG);
      }

      public void debug(Object msg, Throwable ta) {
            logToSap(msg.toString(), ta, Severity.DEBUG);
      }
```



 

```
	public void info(Object msg) {
		logToSap(msg.toString(), null, Severity.INFO);
	}

	...

	private void logToSap(String msg, Throwable ta, int severity) {

		// in case of a unknown severity -> set severity WARNING
		// and add information
		if (severity == -1) {
			severity = Severity.WARNING;
			msg += " (Couldn't identify severity of log4j-Logging
Event!)";
		}

		// log-level DEBUG will be written to default trace
		if (severity == Severity.DEBUG) {
			if (ta == null)
				loc.logT(severity, msg);
			else // for log messages with throwables
				loc.traceThrowableT(severity, msg, ta);
		// everything else will be written to the application.log in
		// the configured category
		} else {
			if (ta == null)
				cat.logT(severity, loc, msg);
			else // for log messages with throwables
				cat.logThrowableT(severity, loc, msg, ta);
		}
	}
}
```

Source code of the logging bridge for JCL.

The source code of a JCL example application is listed below. After requesting a logger for the `JclExampleUsage` class, the various JCL logging methods are called.

```
package org.example.app;

import ...;

/**
 * Does some logging with commons logging API for demonstration.
 */
public class JclExampleUsage {

	/** The JCL-Logger*/
	private static Log jcl = LogFactory.getLog(JclExampleUsage.class);

	/**
	 * Just some commons logging log output.
	 */
	public static void logSomething() {
		// traces
		jcl.trace("JCL - trace-method");
		jcl.debug("JCL - debug-method");
```



 

```
            jcl.debug(
                  "JCL - debug-method with exception",
                  new NullPointerException("Developer's best friend."));

            //logs
            jcl.info("JCL - info-method");
            jcl.warn("JCL - warn-method");
            jcl.error("JCL - error-method");
            jcl.fatal("JCL - fatal-method");
            jcl.fatal(
                  "JCL - fatal-method with exception",
                  new NullPointerException("Developer's best friend."));
      }
}
```

Example application to produce some JCL log messages.

The following JSP script call the `logSomething()` method of our JCL example application.

```
...
<html>
<head>
<title>COMMONS LOGGING to SAP LOGING</title>
</head>
<body>
            <% org.example.app.JclExampleUsage.logSomething(); %>
            <h1>Commons Logging has written to SAP logging.</h1>
            <h1>Check your logs!</h1>
</body>
</html>
```

JSP that calls the example JCL application.

As with the Log4j example, the logs and traces of this example application can be analyzed with the log viewer within the SAP NetWeaver Administrator.

## Run the Examples

The following steps are necessary to run the examples of this tutorial:

1. Download the file `Logging_Tutorial.zip` from SDN.
2. Unzip the content of the zip file into the workspace of the SAP NetWeaver Developer Studio or any other local directory.
3. Download and unzip Log4j (file: `apache-log4j-1.2.16.zip`) from here and Apache Commons Logging (file containing the binaries: `commons-logging-1.1.1-bin.zip`) from here.
4. Copy the files "`apache-log4j-1.2.16\log4j-1.2.16.jar`" and "`commons-logging-1.1.1\commons-logging-1.1.1.jar`" into the folder "`<path to the extracted zip file>\Logging_Tutorial\Logging_Tutorial_Web\lib`".
5. Start the SAP NetWeaver Developer Studio.
6. Switch to the Java EE perspective: "`Window`" -> "`Open Perspective`" -> "`Java EE`".
7. Import the example projects
   a. Select "`File`" -> "`Import`".
   b. Select the import source: "`Other`" -> "`Multiple Existing Projects into Workspace`" and press "`Next`".



 

c. Browse to the folder "Logging_Tutorial". The two subfolders "Logging_Tutorial_Ear" and "Logging_Tutorial_Web" should be listed in the main window.

d. Press "Select All" and "Finish".

8. Build and deploy the imported projects

a. Right mouse click on the newly created project "Logging_Tutorial_Ear" and select "Run As" -> "1 On the Server".

b. In the wizard press "Next" and press "Finish".

c. The deployment may ask for a user and password. Enter "Administrator" and your master password.

9. Test the logging bridges:

a. Start the logging applications in a Web browser:

i. Log4j: http://<host>:<port>/Logging_Tutorial_Web/log4j.jsp

ii. JCL: http://<host>:<port>/ Logging_Tutorial_Web /jcl.jsp

b. Logon to the SAP NetWeaver Administrator: http://<host>:<port>/nwa and select "Log Viewer".

c. Select "Show View" -> "General" -> "SAP Logs (Java)" to see the log entries or "Show View" -> "General" -> "Default Traces (Java)" to see the traces.

10. Pitfalls:

a. If not all log or trace messages appear in the log viewer, set the log and trace levels to "All":

i. From the SAP NetWeaver Administrator start page (http://<host>:<port>/nwa) select "Problem Management" -> "Logs and Traces" -> "Log Configuration" and then "Logging Categories". Change the log level for the category "Applications" -> "MyCategory" to "All" and press "Save Configuration".

ii. To see all traces select "Tracing Locations" within the "Log Configuration" page and then navigate to "org" -> "example" -> "app". Set the log level of the two components "JclExampleUsage" and "Log4jExampleUsage" to "All" and press "Save Configuration".

b. If you use other versions (or more specifically other filenames) than log4j-1.2.16.jar or commons-logging-1.1.1.jar, you need to add the different jar files manually in the build path: In the SAP NetWeaver Developer Studio right click on the "Logging_Tutorial_Web" project and select "Properties" in the context menu. In the new window, choose "Java Build Path" and go to the tab "Libraries". Here you can add alternative versions of the two libraries.

### Use the Logging Bridges in other Java Development Projects

If the examples of this tutorial are used in other Java development projects, the following steps are required to integrate the Log4j framework into SAP's logging framework:

- Copy the files "<path to the extracted zip file>\Logging_Tutorial\Logging_Tutorial_Web\src\com\sap\logging\bridge\log4j\SapLogAppender.java" and "<path to the extracted zip file>\Logging_Tutorial\Logging_Tutorial_Web\WebContent\WEB-INF\classes\log4j.properties" to your source folder.
- Add both files to your class path.
- Set the category name in the log4j.properties configuration file.



 

- Adjust the `SapLogAppender` class to your needs and add it to your class path.

To run the Apache Commons Logging framework:

- Copy the files "`<path to the extracted zip file>\Logging_Tutorial\Logging_Tutorial_Web\src\com\sap\logging\bridge\log4j\ SapLogJclImpl.java`" and "`<path to the extracted zip file>\Logging_Tutorial\Logging_Tutorial_Web\WebContent\WEB-INF\classes\commons-logging.properties`" to your source folder.
- Add both files to your class path.
- Set the category name in the `commons-logging.properties` configuration file.
- Adjust the `SapLogJclImpl.java` class to your needs and add it to your class path.

### Known Limitations

If the logging bridges of this tutorial are used by a Java application, the timestamps for the log entries are set by the SAP logging framework and not by the 3rd party logging frameworks. This means that the timestamps in the log entries contain the times the SAP logging framework has written the logs and not the times when application wrote the logs.



 

## Copyright