**MODULE 7**

**Logging in Tomcat 7**

Logging relevant application details at runtime is a practice that has been followed since the early days of programming. Before debuggers were available, reading log statements was the only way to find out what was going on with an application at runtime. Even with all the debugging tools available today, log statements are still an invaluable tool for monitoring production systems and diagnosing any problems that occur on the deployed application.

In this module, we are going to take a look at the logging options available in Tomcat, for both Tomcat internal logging and logging in individual web application. We will cover following topics:

* Introduce the Java Logging framework and its Tomcat version JULI
* Demonstrate how to use JULI to configure logging in Tomcat
* Introduce Log4j logging framework, its API, and configuration options
* Demonstrate switching to Log4j as Tomcat logging mechanism
* Discuss the usage of the Slf4j framework for web application logging

**Using Tomcat’s JULI Logging Library**

Since version 1.4, Java itself comes with the capable logging package, java.util.logging (Java Logging framework), which enables JVM-wide logging configuration shared by all applications running in the JVM. While it is simple-to-use, customizable, and extensible API, there is one problem with using plain java.util.logging: it’s designed to have a single configuration within the JVM, making it impossible to configure specific logging per-class-loader for separate web applications on Tomcat. In order to get around this issue, Tomcat developers extended the functionality of the standard Java logging framework. The new implementation is known as JULI, and is part of Tomcat’s standard distribution. The package structure has been changed to avoid class-loading issues with JVM-wide logging, and the new code is packaged in tomcat-juli.jar, located in the CATALINA\_HOME/bin directory.

* Note! The access log contains information about all requests handled by Apache Tomcat server. Access logging is handled by Tomcat’s *AccessLogValve*, and is configured in *CATALINA\_HOME/conf/server.xml* file.

**Introduction to Java Logging and JULI libraries**

The main abstractions of the Java Logging framework are loggers—objects that log messages to configured destinations. Loggers are referenced by their configured names. For example, you would typically have one logger per class or package, so that all logging from the given class or package would be performed by a single logger instance; you can use a logger name to reference an existing logger. To instantiate a logger instance in your Java code, you will use static factory method, and construct a logger with the selected name:

*Logger logger = Logger.getLogger("com.apress.mypackage");*

Each logger accepts messages with different logging levels, based on their importance. Using a properties file (as we will demonstrate in the next section), you can configure the level of log output at runtime, decide which logging levels should be written to the destination, and determine which should be discarded. When using Java Logging API, you will invoke different logging method based on the selected level.

Table 7-1 shows the logging levels defined by the Java Logging framework and the API methods used to invoke logging for each level.

***Table 12-1.*** *Logging Levels Defined by Standard the Java Logging Framework*

|  |  |  |
| --- | --- | --- |
| **Level** | **Description** | **API method** |
| *SEVERE* | Used to log exceptions, errors, and other unexpected system failures. | Logger.severe(String message) |
| *WARNING* | Used to log warning messages. | Logger.warning(String message) |
| *INFO* | Used to log information messages, usually in the important steps in the code execution. | Logger.info(String message) |
| *CONFIG* | Logs configuration messages; for example, initialization of objects based on properties files. | Logger.config(String message) |
| *FINE* | Used to log more detailed information; useful for debugging purposes. | Logger.fine(String message) |
| *FINER* | Logs more detailed information than FINE. | Logger.finer(String message) |
| *FINEST* | Logs all messages, to the low-level details. | Logger.finest(String message) |

The levels described are ordered hierarchically, so FINEST is the lowest logging level and it includes all levels above it. For example, if your logging configuration is set to log at FINEST level, it will output all messages logged using *Logger.finest()* method, but also all messages logged using *Logger.finer(), Logger.fine(), Logger.info()*... At the same time, if you configure your logging to SEVERE level, only messages logged using *Logger.severe()* will actually be sent to the output. All other logging calls will be discarded.

Levels below INFO are usually producing huge amount of log information, so it’s not recommended to use them in production. However, they can be very useful during development and testing, or when diagnosing problems in production.

* Note You can specify two more generic options when configuring logging level with JULI: ALL and OFF. ALL has the same meaning as FINEST; all logging levels will be written to the output. OFF disables logging altogether—no logging level will be output.

**Handlers**

Each logger has a list of handlers associated with it (represented by abstract class j*ava.util. logging.Handler*), which are responsible for writing log messages to its destination. There are three main handlers available in Java Logging framework:

* *java.util.logging.ConsoleHandler* simply outputs the logged messages to the JVM System.err output stream.
* *java.util.logging.FileHandler* writes the message to the file on disk, supporting file rotation.
* *java.util.logging.SocketHandler* writes messages to the network socket connection.

Typically, FileHandler is used for most production systems. It’s natural to think of logs as “log-files,” and with a numerous file processing tools available (file editors and log file analyzer) for all platforms, logging to a file has become a standard in application monitoring. While Tomcat’s JULI logging implementation still uses *java.util.logging.ConsoleHandler* and *java.util. logging.SocketHandle*r from Java Logging framework, JULI also provides its own *org.apache.juli.FileHandler*, with improved log buffering capabilities, and has simplified configuration comparing to the one available in JDK as default. We will use JULI implementation of FileHandler in the configuration examples in this chapter.

**Formatters**

The last component of the Java Logging framework that we’re going to discuss is the formatter. There is always one formatter configured for each handler, and its responsibility is to format the log messages before they are sent to output destination. You would typically configure the formatter to include timestamp of the log event, as well as class name and line number where the logging occurred, along with the actual log message. Java Logging library (and Tomcat’s version JULI) comes with the two following formatters: SimpleFormatter and XMLFormatter.

SimpleFormatter formats log message with brief log summary, including date and time of the log

and the logger information. This is the default output format. It writes every log as two lines in the log file:

*Jul 23, 2011 11:23:05 AM org.apache.catalina.core.ApplicationContext log INFO: SessionListener: contextInitialized()*

XMLFormatter writes the log message in XML format (the XML schema for the output format is available on Oracle’s web site <http://download.oracle.com/javase/1.4.2/docs/guide/util/> logging/overview.html#3.0) The following log snippet illustrates an example of an XML formatted log message:

*<record>*

*<date>2011-07-23T11:21:14</date>*

*<millis>1311416474822</millis>*

*<sequence>41</sequence>*

*<logger>org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/examples]</logger>*

*<level>INFO</level>*

*<class>org.apache.catalina.core.ApplicationContext</class>*

*<method>log</method>*

*<thread>11</thread>*

*<message>SessionListener: contextInitialized()</message>*

*</record>*

In addition to these two, JULI logging implementation adds three convenient formatters (located in the org.apache.juli package):

* OneLineFormatter has the same format as the default SimpleFormatter, but written in a single line, so it’s easier to process.
* VerbatimFormatter writes the log message only, without any additional information.
* JdkLoggerFormatter uses a compact output format, with a timestamp in the millisecond format and abbreviated log level:

1311416945846 I [/examples]ContextListener: contextInitialized()

The human-readable SimpleFormatter is usually sufficient for most scenarios. If you require searching and processing of log messages, OneLineFormatter can be a good alternative, as it’s easier to grep log files if they contain end-of-line separator after each message.

**Logging Configuration**

The logging configuration is typically specified in the file logging.properties. Listing 12-1 shows the simple logging configuration.

***Listing 7-1.*** *Java Logging Configuration Is Provided in the logging.properties Configuration File*

*handlers = 1myhandler.org.apache.juli.FileHandler, ‘2anotherhandler.org.apache.juli.FileHandler #1*

*.handlers = 1myhandler.org.apache.juli.FileHandler #2*

*1myhandler.org.apache.juli.FileHandler.level = FINE #3 1myhandler.org.apache.juli.FileHandler.directory = ${catalina.base}/logs #4 1myhandler.org.apache.juli.FileHandler.prefix = foo. #5 1myhandler.org.apache.juli.FileHandler.formatter= org.apache.juli.OneLineFormatter #6*

*2anotherhandler.org.apache.juli.FileHandler.level = FINE*

*2anotherhandler.org.apache.juli.FileHandler.directory = /var/logs #7 2anotherhandler.org.apache.juli.FileHandler.prefix = bar.*

*com.mycompany.app1.MyClass.handlers=2anotherhandler #8*

*com.mycompany.app1.MyClass.level=INFO #9*

The first line of configuration defines all handlers that we are going to use (#1). In the standard Java Logging framework, each handler is specified by its class name, as there is only one instance of each handler in JVM. As we mentioned before, Tomcat’s JULI implementation supports handler-per-class- loader configuration, and that is reflected in the configuration by adding a prefix to each handler type. A handler prefix starts with a number, then has an arbitrary string, and ends with a period (.). In our sample, we define two file handlers, prefixed with 1myhandler and 2anotherhandler, respectively.

Next, we specify the default handlers using .handlers property (#2). The default handlers will be used for loggers that do not have a specific handler configured. We specify 1myhandler as default handler.

Now we need to configure properties for each of the handlers:

Default logging level for handler is specified using level property (#3). When you configure loggers, you can override this property for each logger, but you cannot set it to lower level than specified here. In our example, we set the default level to FINE, which means we can logger configuration can override it to CONFIG or INFO, but not to FINER or FINEST level.

Next, we set the directory where the log files should be stored (#4). You can use Tomcat’s environment properties in logging configuration: we configured the logs directory relative to CATALINA\_HOME.

The prefix property sets the filename prefix to use when creation log files (#5).

Similarly to prefix, you can set the file suffix, but if you don’t it will default to .log.

Finally, we set the formatter for 1myhandler handler to already described OneLineFormatter (#6).

The same configuration is required for the second handler. In this case we used the absolute directory path (#7), and omitted the formatter configuration, which means the default *SimpleFormatter* will be used.

Finally, we need to configure loggers for the defined handlers:

The *1myhandler* is defined as default one, so we don’t need to specify logger for it— all logs to non-existing loggers will go to this handler.

We define one logger, com.mycompany.app1.MyClass, and set its handler property to *2anotherhandler* (#8).

We also override its log level to INFO (#9). We can do this, as INFO is higher in the hierarchy of levels then the default handler level FINE.

We configured the loggers, but how can we use them? In the following paragraphs, we are going to explain a few example code snippets demonstrating the Java Logging API.

To instantiate a logger with given name, you can use factory method in the java.util.logging.Logger class:

*Logger logger = Logger.getLogger("com.mycompany.app1.MyClass");*

You can now log messages to the logger.

**Example 1:**

*logger.info("INFO MESSAGE");*

In the previous example, Java Logging framework will match the logger name to already configured logger (“com.mycompany.app1.MyClass”). Because it’s configured to accept INFO logging level, 2anotherhandler will write the log message to the /var/logs/bar.log file.

**Example 2:**

*logger.warning("WARNING MESSAGE");*

This message will also be written to the /var/logs/bar.log file, because WARNING log level is higher in the level hierarchy then the level configured for this logger (INFO).

**Example 3:**

*logger.fine("TRACE MESSAGE");*

Although *2anotherhandler* will accept logging on FINE level, this message will not be visible in the log. This is because the “*com.mycompany.app1.MyClass*” logger overrides the logging level to INFO, and since the FINE level is lower than configured level, the log will not be written.

Let’s take a look at another logger:

*Logger logger2 = Logger.getLogger("testlogger");*

There is no logger configured with name “*testlogger*,” so the root logger will be used, with default handler (*1myhandler*).

**Example 4:**

*logger2.info("INFO MESSAGE");*

This message will be written to the CATALINA\_HOME/logs/foo.log file, as configured for 1myhandler.

**Example 5:**

*logger2.fine("TRACE MESSAGE");*

This message will be written to the CATALINA\_HOME/logs/foo.log file, as configured for 1myhandler. Because the logger with given name does not exists, the level configure on the default handler will be used.

**Example 6:**

*Logger2.finer("DETAILED TRACE MESSAGE");*

The level FINER is lower than the FINE level configured for default handler, so this message won’t be written to the log file

**Rotating Logs**

You can use Tomcat JULI logging framework to rotate log files daily. The format of the log files is {prefix}{date}{suffix}. The new file is created at midnight every day, and used for logging until end of that day.

The log file rotation is enabled by default, but you can control it using the rotatable property in your handler configuration:

*1myhandler.org.apache.juli.FileHandler.rotatable = false*

The rotation can only rotate files daily, and you don’t have any options about the filename format used. If you need richer rotating options (rotating by maximum file size, weekly, or monthly, for example), use java.util.logging.FileHandler or some of the specialized logging frameworks, like Log4j, which we will discuss in the second part of this chapter.

**Servlet API Logging**

The Java Servlet specification defines the logging API to be used when logging messages from servlet-related classes. The logging based on Servlet API is performed by calls to ServletContext.log(String message) method, passing in the message to be written to the log.

This logging method has become obsolete with emergence of more powerful, customizable logging frameworks for Java, such are java.util.logging or log4j. Developers tend to prefer these logging frameworks to Servlet API logging.

In Tomcat, all messages logged to Servlet log are intercepted and handled by internal Tomcat logging. Tomcat provides handlers for ServletContext logs, for each engine, host, and context combination. The names of such loggers follow the convention:

*org.apache.catalina.core.ContainerBase.[ENGINE].[HOST].[CONTEXT]*

All ServletContext loggers’ names start with org.apache.catalina.core.ContainerBase, followed by the engine name, host name, and context name in square brackets, separated by dots (.). The following code snippet illustrates the logger configuration:

*org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].level = INFO org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].handlers =3manager.org.apache.juli.FileHandler*

Now that we covered the basic concepts of Java Logging framework and Tomcat-specific improvements in JULI, we can have a look at how to configure Tomcat to use it. In the next two sections, we cover the configuration of internal Tomcat logging and logging in web applications deployed to Tomcat.

**Configuring Internal Tomcat Logging with JULI**

The server-wide logging configuration is located in the CATALINA\_HOME/conf/logging. properties file. We’re going to explain the internal Tomcat logging configuration using the default logging.properties file, which is part of the Tomcat distribution. Listing 7-2 shows the contents of the default logging.properties file.

***Listing 12-2.*** *Default JULI Logging Configuration in the logging.properties file*

handlers = 1catalina.org.apache.juli.FileHandler, 2localhost.org.apache.juli.FileHandler, 3manager.org.apache.juli.FileHandler, 4host-manager.org.apache.juli.FileHandler, java.util.logging.ConsoleHandler #1

.handlers= 1catalina.org.apache.juli.FileHandler, java.util.logging.ConsoleHandler #2

1catalina.org.apache.juli.FileHandler.level = FINE #3 1catalina.org.apache.juli.FileHandler.directory = ${catalina.base}/logs #4 1catalina.org.apache.juli.FileHandler.prefix = catalina. #5

2localhost.org.apache.juli.FileHandler.level = FINE 2localhost.org.apache.juli.FileHandler.directory = ${catalina.base}/logs 2localhost.org.apache.juli.FileHandler.prefix = localhost.

#Similar configuration is available for manager and host-manager web application.

#which is omitted here for clarity

java.util.logging.ConsoleHandler.level = FINE java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].level = INFO org.apache.catalina.core.ContainerBase.[Catalina].[localhost].handlers = 2localhost.org.apache.juli.FileHandler

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].level = INFO org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].handlers =3manager.org.apache.juli.FileHandler

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/host-manager].level = INFO org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/host-manager].handlers = 4host-manager.org.apache.juli.FileHandler

Tomcat by default defines five log handlers: one system wide ConsoleHandler and four FileHandlers (#1). Both the 1catalina FileHandler and the ConsoleHandler are configured as default handlers (#2); all messages logged to non-configured loggers will go to both of these handlers.

Each of the five handlers is configured separately. Log level is set to FINE for all of them by default, so no messages logged at FINER or FINEST level will appear in any tomcat logs (#3). All FileHandlers are storing log files to CATALINA\_HOME/logs directory by default (#4).

There are four different file handlers configured. Handler 1catalina is configured to write log messages to CATALINA\_HOME/logs/catalina.log file (#5). Because of the file rotation, which is enabled by default, the file name will actually contain the date as well—for example, catalina.2011-07-21.log.

Similar configuration (with different filename prefixes only) is defined for other three file handlers.

Because ConsoleHandler and 1catalina FileHandler are configured as default log handlers, there are no loggers configured for them.

The remaining three file handlers each have one logger configured. These loggers are

*org.apache.catalina.core.ContainerBase.[Catalina].[localhost]* logs all *ServletContext.log*() messages for all web applications deployed to host with name “*localhost*”, which is the default host available in Tomcat. The handler for this logger is 2localhost, which writes to *CATALINA\_HOME/log/localhost.log*.

*org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager]* logs all ServletContext.log() messages for the Tomcat’s Manager web application. The handler for this logger is 3manager, which writes to CATALINA\_HOME/log/manager.log.

*org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/host-manager]* logs all ServletContext.log() messages for the Tomcat’s Manager web application. The handler for this logger is 4host-manager, which writes to the *CATALINA\_HOME/log/host-manager.log* file.

**Console Logging**

The console output in Tomcat contains all messages sent to System.out and System.err output streams, and all exception stack traces. On Linux systems, and when running Tomcat as a windows service, Tomcat intercepts all console output and logs it to the file configured by CATALINA\_OUT environment variable. By default, the location of this file is CATALINA\_HOME/logs/catalina.out on Linux-based systems. If you’re running Tomcat as a Windows service, console output will be appended to files CATALINA\_HOME/logs/stdout.log and CATALINA\_OUT/logs/stderr.out instead.

While it is useful to have console output in a file, there are few pitfalls that you should be aware of. On Linux, the console output file (catalina.out) cannot be rotated without stopping Tomcat. If you have a lot of System.out output in your applications, these files can grow huge, and affect the Tomcat performance and stability.

For production environments, it’s advisable not to use console output at all, but to log everything using chosen logging framework. You can disable console logging in Tomcat by removing ConsoleHandler from logging configuration. If you want to disable console logging for selected web applications only, you can set the swallowOutput attribute to true in the <Context> element configuration for desired web application:

*<Context path="/manager" swallowOutput="true">*

But make sure you have proper logging configured before you do this. If the console output is the only logging you have configured, you will not see any logging from your web application.

**Configuring Web Application Logging with JULI**

To configure JULI for each web application, all you need to do is create web-application specific logging.properties file and package it in WEB-INF/classes directory of your web application. Listing 7-3 illustrates a simple web application logging configuration.

***Listing 7-3.*** *Single Web Application Logging Configuration*

*handlers=1chapter7.org.apache.juli.FileHandler, 2chapter7.org.apache.juli.FileHandler #1*

*handlers=1chapter7.org.apache.juli.FileHandler #2*

*1chapter7.org.apache.juli.FileHandler.level=FINE 1chapter7.org.apache.juli.FileHandler.directory=${catalina.base}/logs*

*1chapter7.org.apache.juli.FileHandler.prefix=chapter7-default. #3*

*2chapter7.org.apache.juli.FileHandler.level=FINE 2chapter7.org.apache.juli.FileHandler.directory=${catalina.base}/logs 2chapter7.org.apache.juli.FileHandler.prefix=chapter7-special. #4*

*com.apress.apachetomcat7.chapter7.level=INFO*

*com.apress.apachetomcat7.chapter7.handlers=*

*2chapter7.org.apache.juli.FileHandler #5*

If the Tomcat’s class loader detects the logging.properties file located in the WEB-INF/classes directory of the web application, it will use it to configure logging for that web application—ignoring the configuration in CATALINA\_HOME/conf/logging.properties. This means that no default handlers from Tomcat’s main logging configuration will be defined, and you should take care to add default handlers in your web application’s configuration, or you risk losing logged messages.

We defined two file logging handlers, 1chapter7 and 2chapter7 (#1). We are going to use the 1chapter7 as default handler (#2), so any loggers not explicitly defined will write output to this handler. We configured 1chapter12 handler to log to file chapter12-default.log, located in the CATALINA\_HOME/logs directory (#3).

We will use 2chapter7 handler for “special” web application logging, and it will log messages to the chapter7-special.log file (#4).

Finally, we created a single logger, which uses 2chapter12 handler, and gave it a name to match our web application’s root package: com.apress.apachetomcat7.chapter12.

Now in the servlet Java code we implemented, we can simply log to the configured logger:

*Logger logger = Logger.getLogger("com.apress.apachetomcat7.chapter7"); logger.warning("test message");*

And the message will appear in the CATALINA\_HOME/logs/chapter7-special.log file:

*Jul 23, 2011 5:04:45 PM com.apress.apachetomcat7.chapter12.HelloWorldServlet doGet WARNING: test message*

If you log your message to a logger that isn’t configured in the logging.properties file, the message will be appended to the file configured as the default handler:

*Logger logger2 = Logger.getLogger("unknown logger"); logger2.info("This message will go to console");*

The message will not be present in the chapter12-special.log, but instead you can see it in the chapter7-default.log file:

*Jul 23, 2011 5:35:22 PM com.apress.apachetomcat7.chapter12.HelloWorldServlet doGet INFO: This message will go to the console output*

Using this approach, you can configure as many loggers as you need for your web application.

Common strategy is to have different layers of the web application log to separate files (so you can

have web.log, services.log, dataaccess.log). Or, for smaller web applications, you can have a single file handler defined as default handler, and log all messages to a single file.

* Note You will see no output in the Tomcat console in case you handle all logging using Java logging properties. You will have to check the configured log file in order to see output from your application. But you won’t lose any messages, so that can only be a good thing.

By using a logging library, such as Java logging or its JULI version, you will get consistent log handling throughout your web application, without fear that some messages will be lost.

Although JULI logging can address all your application’s needs for logging, there are other logging frameworks that gained popularity before Sun included Java Logging framework with Java 1.4. One of the most popular logging frameworks is Apache’s Log4j, and open source logging implementation providing easy-to-use API, rich configuration options, and high performance. In the next section, we are going to see how we can use Log4j instead of JULI framework for Tomcat logging.

**Using Log4j Library for Web Application Logging**

Sun has introduced Java Logging framework from Java version 1.4. However, the need for a powerful logging framework in Java was present long before that. In addition, the Java Logging framework was only available for Java SDK 1.4 and later, so applications running on Java 1.3 and earlier could not use it.

Ceki Gulcu originally wrote Log4j, and it quickly gained popularity, becoming an open source project as part of the Apache Software Foundation. With a large user base, Log4j is still one of the most widely used logging frameworks, and a lot of the features in Java Logging framework features were inspired by Log4j.

Just like with the Java Logging framework, logger is a central abstraction in Log4j. However, the logger in Log4j is defined as class org.apache.log4j.Logger (as opposed to java.util.logging.Logger in Java Logging). Log messages are handled by appenders, which have the same role as java.util.logging handlers. Finally, messages are formatted using layouts in Log4j, which have the same role as formatters in Java Logging framework. Table 7-2 shows the comparison between corresponding elements from Java Logging framework and Log4j.

***Table 7-2.*** *Main Components of Java Logging Framework and Log4J*

|  |  |
| --- | --- |
| **Java Logging** | **Log4j** |
| *java.util.logging.Logger* | org.apache.log4j.Logger |
| *java.util.logging.Handler* | org.apache.log4j.Appender |
| *java.util.logging.Formatter* | org.apache.log4j.Layout |

Log4j uses slightly different logging levels than Java Logging framework. Because Log4j was started before Sun’s Java Logging, developers are usually more familiar with logging levels from Log4j. Tomcat’s JULI implementation even offers abstraction of java.util.logging.Logger that uses the Log4j levels instead of Java Logging standard levels. Table 7-3 lists the logging levels in Log4j.

***Table 7-3.*** *Logging Levels and Their API Log Calls in Log4J*

|  |  |  |
| --- | --- | --- |
| **Level** | **Description** | **API Method** |
| *FATAL* | Used to log fatal errors, when an application cannot recover and continue. | Logger.fatal(String message) |
| *ERROR* | Used to log error messages; for example, handled exceptions. | Logger.error(String message) |
| *WARN* | Used to log warnings. | Logger.warning(String message) |
| *INFO* | Logs information messages. | Logger.info(String message) |
| *DEBUG* | Logs developer debug information, usually used during development and diagnostics. | Logger.debug(String message) |
| *TRACE* | Traces information about application execution. | Logger.trace(String message) |

* Note DEBUG and TRACE logging levels log huge amounts of information, and they should be turned off in production systems. They are analogous to FINE, FINER, and FINEST log levels in JULI.

In order to use Log4j, you have to place log4j-{version}.jar file on your application classpath. The configuration can be provided using the properties file (log4j.properties) or as XML file (log4j.xml). In this module we will be using properties file for Log4j configuration. The configuration file needs to be located on the application classpath as well. Listing 7-4 illustrates the simple Log4j configuration file.

***Listing 7-4.*** *Simple Log4j Configuration File*

*log4j.rootLogger=DEBUG, FOO #1*

*log4j.appender.FOO =org.apache.log4j.ConsoleAppender #2*

*log4j.appender.FOO.layout=org.apache.log4j.PatternLayout #3 log4j.appender.FOO.layout.ConversionPattern=%d [%t] %-5p %c - %m%n #4 log4j.logger.com.foo=WARN #5*

All configuration elements must start with string log4j. In Log4j, it’s mandatory to configure root logger (#1), which will take over the logging for all loggers that are not explicitly configured (similar to default handlers in Java Logging). The name of the logger is the property name (log4j.logger), and the value contains logging level (DEBUG) and the name of the appender to use (FOO).

Next, we configure the appender, using Log4j’s ConsoleAppender (#2), which appends all log messages to console output (similar as Java Logging’s ConsoleHandler). We set the name of the appender to FOO.

The next part is the log layout (#3). We are using PatternLayout, a powerful pattern formatter from Log4j. The pattern to be used with PatternLayout is specified as separate configuration property, ConversionPattern (#4).

* Note Although Log4j supports a few other layouts, with the power of PatternLayout you will probably not need any other. The configuration options of PatternLayout itself are a good reason to start using Log4j for logging. We will discuss features of PatternLayout in the separate section.

Finally, we specify another logger, called com.foo, and set its logging level to WARN (#5). Note that we haven’t specified the appender for com.foo logger—Log4j will inherit the appender from root logger in this case.

Logger inheritance is mandatory in Log4j and is a very convenient mechanism for logging

configuration. Logger com.foo will accept log messages from loggers com.foo, but also com.foo.bar, com.foo.bar.MyClass, com.foo.bar.MyClass$Test... Using logger inheritance mechanism, you can easily configure specific loggers on package levels in your application, and use class name to reference underlying logger:

**Listing 7-4.**

*package com.foo.bar*

*import org.apache.log4j.Logger; public class MyClass{*

*Logger logger = Logger.getLogger(MyClass.class.getName());*

*…*

*logger.warn ("Warning message");*

*…*

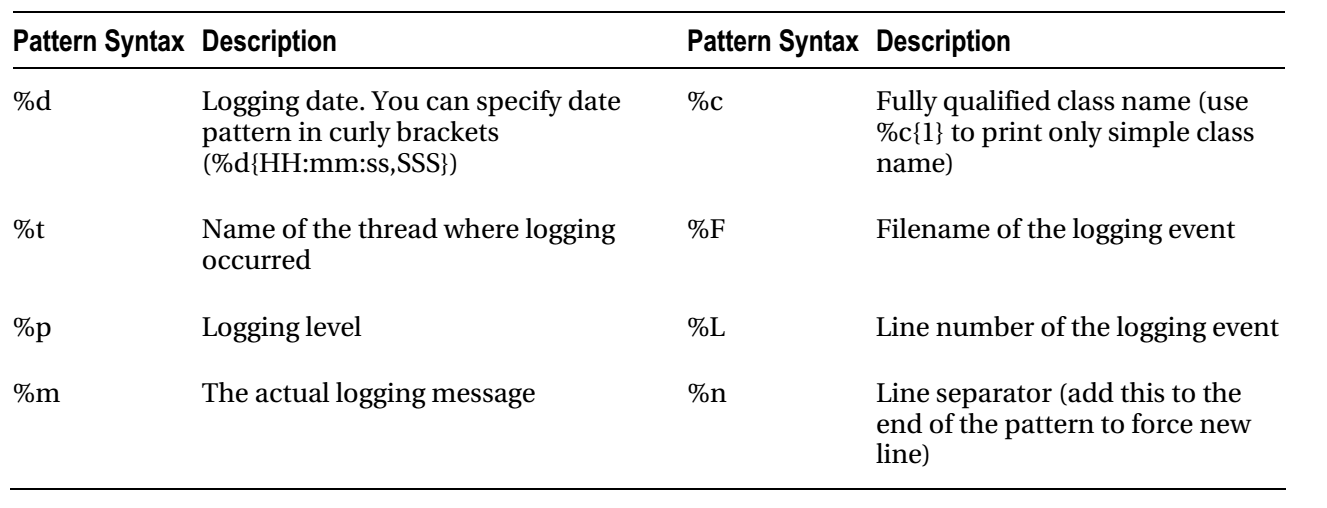
*}*

The logger instantiation in the previous code snippet is the same as Logger.getLogger ("*com.foo.bar.MyClass*"), and in Log4j this logger will inherit configuration of logger *com.foo* from Listing 7-4.

**PatternLayout**

PatternLayout is a flexible and powerful class designed to transform logging events to text in Log4j, based on the supplied conversion pattern. The conversion pattern options are largely inspired by the printf function in C programming language. Table 7-4 lists some of the commonly used pattern configuration options.

Table 7-4. Common Pattern Configuration Elements for Log4j’s PatternLayout



You can specify text width and justification for every pattern element:

* %20m: Specify minimum width of 20 characters.
* %20.50m: Specify minimum and maximum width to 20 and 50 characters, respectively.
* %-20m: Specify minimum width of 20 characters, left justified.

Here are some of the pattern examples and the log entries they generated:

*%d [%t] %-5p %c - %m%n*

* 1. *14:07:41,508 [main] INFO MyClass – Executing...*

*%5p [%t] (%F:%L) - %m%n*

*INFO [main] (MyClass.java:12) – Executing...*

Now that we covered basic functionality and configuration of Log4j, let’s take a look how we can configure Tomcat and web applications to use Log4j logging instead of JULI.

**Using Log4j for Tomcat Internal Logging**

When you download Tomcat, JULI logging library and configuration are available out of the box. In order to use Log4j for Tomcat internal logging instead, you will need to replace the existing JULI library with the Log4j-JULI integration. You can achieve that by following these six steps:

1. Delete existing JULI library (CATALINA\_HOME/bin/tomcat-juli.jar file) and the existing Tomcat Java Logging configuration file (CATALINA\_HOME/conf/logging.properties).
2. Download JULI Log4j Tomcat library (tomcat-juli.jar) from the Tomcat downloads’ Extras section ([http://tomcat.apache.org/download-70.cgi).](http://tomcat.apache.org/download-70.cgi)) Place the downloaded file to CATALINA\_HOME/bin directory.
3. Download Tomcat JULI adapters library (tomcat-juli-adapters.jar) from the Tomcat downloads’ Extras section. Place this file in the CATALINA\_HOME/lib directory.
4. Download Log4j (version 1.2 or later), and place the downloaded library file to CATALINA\_HOME/lib directory.
5. Create the Log4j configuration file at the following location: CATALINA\_HOME/lib/log4j.properties.

Listing 7-5 illustrates the log4j configuration matching the default Java Logging configuration.

1. Restart Tomcat.

***Listing 7-5.*** *The Contents of the Log4j configuration File Matching the Default Tomcat Logging Settings*

log4j.rootLogger=INFO, CATALINA

# Define all the appenders log4j.appender.CATALINA=org.apache.log4j.DailyRollingFileAppender log4j.appender.CATALINA.File=${catalina.base}/logs/catalina. log4j.appender.CATALINA.Append=true log4j.appender.CATALINA.Encoding=UTF-8

# Roll-over the log once per day log4j.appender.CATALINA.DatePattern='.'yyyy-MM-dd'.log' log4j.appender.CATALINA.layout = org.apache.log4j.PatternLayout

log4j.appender.CATALINA.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

log4j.appender.LOCALHOST=org.apache.log4j.DailyRollingFileAppender log4j.appender.LOCALHOST.File=${catalina.base}/logs/localhost. log4j.appender.LOCALHOST.Append=true log4j.appender.LOCALHOST.Encoding=UTF-8 log4j.appender.LOCALHOST.DatePattern='.'yyyy-MM-dd'.log' log4j.appender.LOCALHOST.layout = org.apache.log4j.PatternLayout log4j.appender.LOCALHOST.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

log4j.appender.MANAGER=org.apache.log4j.DailyRollingFileAppender log4j.appender.MANAGER.File=${catalina.base}/logs/manager. log4j.appender.MANAGER.Append=true log4j.appender.MANAGER.Encoding=UTF-8 log4j.appender.MANAGER.DatePattern='.'yyyy-MM-dd'.log' log4j.appender.MANAGER.layout = org.apache.log4j.PatternLayout log4j.appender.MANAGER.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

log4j.appender.HOST-MANAGER=org.apache.log4j.DailyRollingFileAppender log4j.appender.HOST-MANAGER.File=${catalina.base}/logs/host-manager. log4j.appender.HOST-MANAGER.Append=true log4j.appender.HOST-MANAGER.Encoding=UTF-8 log4j.appender.HOST-MANAGER.DatePattern='.'yyyy-MM-dd'.log'

log4j.appender.HOST-MANAGER.layout = org.apache.log4j.PatternLayout log4j.appender.HOST-MANAGER.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender log4j.appender.CONSOLE.Encoding=UTF-8 log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout log4j.appender.CONSOLE.layout.ConversionPattern = %d [%t] %-5p %c- %m%n

# Configure which loggers log to which appenders log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost]=INFO, LOCALHOST log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager]=INFO,MANAGER

log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/host-manager]= INFO, HOST-MANAGER

And that’s it: your Tomcat will now use Log4j for all internal logging. For additional information and details about the Log4j configuration, take a look at the Log4j project home page <http://logging.apacheorg/log4j/index.html>.

**Using Log4j for Web Application Logging**

In order to have your application use Log4j for logging, you’ll have to make to add log4j.jar library to the web application classpath (in the WEB-INF/lib directory of your web application). Then, you’ll have to place Log4j configuration file, log4j.properties, into the web application classpath as well (in the WEB- INF/classes directory). Listing 7-6 shows sample log4j.properties file you can use.

***Listing 7-6.*** *Sample Log4j Configuration for a Web Application Using RollingFileAppender*

log4j.rootLogger=WARN, rolling #1

log4j.appender.rolling=org.apache.log4j.RollingFileAppender #2

log4j.appender.rolling.File=/var/log/default.log #3

log4j.appender.rolling.MaxFileSize=10MB #4

log4j.appender.rolling.MaxBackupIndex=5 #5

log4j.appender.rolling.layout=org.apache.log4j.PatternLayout #6 log4j.appender.rolling.layout.ConversionPattern=%p [%t] %C{1}.%M(%L) | %m%n

log4j.logger.com.apress.apachetomcat7.chapter12=DEBUG #7

log4j.logger.javax.servlet=INFO #8

The root logger is configured to log WARN logging level, and use appender with name rolling (#1). Next, we configure the rolling appender—we’re using Log4j’s RollingFileAppender in this example (#2). RollingFileAppender writes output to a file configured using File property (#3). When the log file size reaches size specified using MaxFileSize property—10MB, in our example (#4)—the RollingFileAppender will create a new file and continue logging to it. We specify the number of old log file to keep as backup using MaxBackupIndex property (#5).

Finally, we configure layout for the rolling appender—we’re using familiar PatternLayout (#6), with ConversionPattern set. We added two additional loggers, extending the root logger. First we added a logger for the classes in the main package of our web application, overriding logging level to DEBUG (#7). Second logger we added logs any messages from javax.servlet package with INFO level (#8). You can use this sample configuration, and the earlier Log4j configuration examples to create Log4j configuration for most common usage scenarios. For more information about Log4j, consult Log4j documentation ([http://logging.apache.org/log4j/1.2/publications.html).](http://logging.apache.org/log4j/1.2/publications.html))

**Web Application Logging Using Slf4j Library**

Both Log4j and Java Logging framework provide logging functionality that can be used in any Java project. The biggest issue in choosing one or another is that you cannot switch easily. Not only that these frameworks use different libraries and configuration, but they have incompatible APIs (java.util.logging.Logger vs. org.apache.log4j.Logger), so you would have to replace every logging call in your code from one framework to another. This can be a tedious job, and most of the times developers stick to the first choice logging library for the entire life of the project.

This is unfortunate, as the differences between frameworks aren’t that big after all. How good would it be to have a bridging library, with unified API that can pick any logging implementation that is available on the classpath? Welcome Simple Logging Façade for Java (commonly called Slf4j), Java logging bridge library designed exactly for that reason.

* Note Apache commons-logging library was designed with the same goal, as a universal logging API that delegates logging to the discoverable underlying logging framework. However, due to class-loading issues with the commons-logging discovery mechanism, it is recommended to avoid it and use Slf4j instead.

**Using Slf4j**

To start using Slf4j, you will need slf4j-api.jar library, which contains the Slf4j API classes and interfaces.

* Note You can download slf4j-api.jar and all other slf4j libraries from the project’s download page: [www.slf4j.org/download.html.](http://www.slf4j.org/download.html)

Once you have slf4j-api.jar on your project’s class path, you can use its API to log statements in your Java code. Listing 12-7 demonstrates its usage.

***Listing 12-7.*** *Using Slf4j API for Logging in Java*

import org.slf4j.Logger; #1

import org.slf4j.LoggerFactory; #2

…

Logger logger = LoggerFactory.getLogger(HelloWorld.class); #3 logger.info("Hello World"); #4

logger.warn("Warning message"); #5

Note that we imported the Slf4j classes in this case (#1, #2), instead of logging-framework specific ones. We will use org.slf4j.Logger for all logging (instead of *java.util.logging.Logger* or org.apache.log4j.Logger, which we used before). The Logger instance is created using LoggerFactory static factory method (#3). You can use the Slf4j Logger in the familiar manner, to log message on INFO level (#4), WARN level (#5), or any other logging level available.

Note You will notice that the Slf4j API is more similar to Log4j than to Java Logging, and that it uses Log4j logging levels as well. This is understandable as the creators of Log4j started the Slf4j project.

You can use the previously mentioned API calls in your code regardless of which underlying logging framework you decide to use—Slf4j will bridge these calls to the native API of logging library used. Slf4j achieves this by using bindings, the Java library that bridges the Slf4j API calls to the native logging calls. Slf4j has bindings for all popular logging frameworks, including Java

Logging framework and Log4j, which we used throughout this chapter. All you have to do is download the binding for your logging library, and add it to the application classpath. Of course, you will have to include the logging library itself and its configuration as well.

Let’s see how would this work with Java Logging library, for example. You would include Java Logging library on your classpath, including its configuration in *logging.properties* file—just like we described in the first part of this chapter. You will also include slf4j-api.jar on your library with the Java Logging binding (slf4j-jdk14.jar), which can be downloaded from the Slf4j download page. To actually log messages in your code, you will use slf4j API (*org.slf4j.Logger* and *org.slf4j.Loggerfactory* classes). When you deploy your application to Tomcat, the logging will work just the same as before. Invisible to you, the org.slf4j.Logger calls will be bridged to the *java.util.logging.Logger* class, using the binding library supplied on classpath.

Let’s say that you decide to switch to Log4j a few months later. You will have to remove Java Logging configuration from your application classpath, and include Log4j library and log4j.properties configuration file instead. You will also have to remove the Java Logging Slf4j binding (slf4j-jdk14.jar), and replace it with the binding for Log4j (slf4j-log4j.jar).

And that’s it! There is no need to change any of the API calls, as Slf4j API you used will still work.

Only this time it will find the log4j binding on the classpath, and it will bridge all calls to org.slf4j.Logger class to the Log4j’s org.apache.log4j.Logger, as expected!

In case you tried to switch logging framework without using Slf4j in the first place, you would need to change all source files where you used *java.util.logging* classes, replace it with log4j API, recompile, repackage and redeploy the application—a time consuming, error-prone exercise.

Slf4j enables you to choose logging framework at deployment time, and change it as often as you like, using only classpath changes (adding and removing jar files and configuration). It supports all major logging frameworks, and it has rich API and excellent performance. If you’re starting new project, you should consider using Slf4j as a bridge to your selected logging framework.

## **Summary**

In this chapter we discussed the logging options available in Apache Tomcat. We configured default JULI logging for Tomcat internal logging and for logging in individual web applications deployed in Tomcat.

We then introduced Log4j, a popular logging library, and explained how you can switch to Log4j for Tomcat internal logging. We also demonstrated how you can use Log4j for web application logging.

Finally, we demonstrated how you can use Slf4j API and bindings to abstract the concrete logging library in use, and enable easy switching between logging frameworks at deployment time.

In the next chapter, we are going to take a look at JNDI resources in Tomcat and use JNDI to configure database connections and mail sessions.