Sakai Developer Logging

March 23, 2005

Please direct questions or comments about this document to:
Glenn R. Golden, ggolden@umich.edu

Code developed for Sakai, like code developed everywhere else, often has the need to log messages at runtime that capture normal and unusual and error programming conditions. These messages usually are sent to a log file or some other destination.

Logging Approaches

There are two different approaches to logging generally used in Java. One is exemplified by the popular log4J package, Apache Commons Logging, and the built-in logging provided by Java. In this approach, each java class declares a logging object from the selected package as a class member variable, and uses this to log messages.

The other approach uses an API programming paradigm. In this approach, there's a logging API and some sort of implementation component. Our programs then inject the component or discover the component at runtime and call on it to do logging.

Logging In Sakai

The Sakai framework supports three different types of logging. The Sakai Log API provides a LogManager that can be injected, discovered, or called via cover to log messages. The LogManager API is similar in design to Commons Logging.

Code in Sakai can also set up a Commons Logging `Log` object in each class, and use that for logging.

Sakai code also can set up a log4j `Logger` object in each class, and use that for logging.

All three logging approaches end up in the same log file.

Sakai Log API

Sakai provides a LogManager in the Log API. This API is designed to be compatible with the concepts of the Apache Commons Logging class, and is also similar to the built-in Java logger and log4j.

These capabilities include the ability to log a string message, or a string message and a throwable, at any of these logging levels:

- trace
- debug
- info

- warn
- error
- fatal

These levels are used to limit the logging output. The output is configured to a particular logging level. Messages at this level or "above" (later in the list) are logged – those "below" are not.

The API also supports the testing of logging level set. This is important to limit the runtime work to compute logging messages for log levels not currently in use. Best practice is to guard any logging with a test, as in this example:

```
if (m_logger.isDebugEnabled())
{
        m_logger.debug(message);
}
```

Guarded logging should be practiced for any level below "warn", since we can assume that warnings will always be enabled even in our production systems. This same practice is supported by all three logging options.

The Log API is in the `SAF` module `log` project.

Sakai Log Component

The standard implementation of the Log API shipped with Sakai is the framework component `CommonsLogComponent`. This implements the LogManager using Apache Commons Logging calls. All log messages that are created by the LogManager end up being sent to a single Commons Log object, called:

```
org.sakaiproject.component.framework.log.CommonsLogComponent
```

This name is important when you configure the logging for your server.

The Log Component is in the `SAF` module `log-component` project.

Commons Logging

Apache Commons Logging (http://jakarta.apache.org/commons/logging/) can also be used in any Sakai code. The code is already installed and configured by the Sakai framework; you need only use it in your components and tools.

See their website for detailed documentation about how to use commons logging. In very short, add this code to your class to have a log:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class CLASS
```

```
	{
		/** Our log (commons). */
		private static Log M_log = LogFactory.getLog(CLASS.class);
```

The log is named with the package and class name of your class.

Because Sakai sets up log4j, commons logging will use log4j to implement logging. Tomcat will for the most part also use log4j for its logging – only parts of startup and shutdown come into the logs not under log4j's control.

Log4j Logging

Apache also defines the log4j project (http://logging.apache.org/log4j/). Sakai installs and configured log4j, so it too is available to be used.

See their website for detailed documentation about how to use log4j logging. In very short, add this code to your class to have a logger:

```
	import org.apache.log4j.Logger;

	public class CLASS
	{
		/** Our logger (log4j). */
		private static Logger M_log =
			Logger.getLogger(CLASS.class);
```

The log is named with the package and class name of your class.

Log Configuration

All the logging from Sakai, through the LogManager API, Commons Logging or log4j, end up using log4j and a log4j configuration. The `SAF` module `log-configure` project includes a `log4j.properties` configured for Sakai. This becomes a jar file that is deployed to `/common/lib/` (the `component` project is responsible for deploying Commons Logging to `/shared/lib/`, and log4j to `/common/lib/`).

This configuration file has options to choose to send all the logging to standard out, which in Tomcat ends up in `catalina.out`, or to a rotating log file. See the log4j documentation for all the configuration options available.

This configuration file can also be used to set the logging levels for the entire log output, or for specific loggers.

Here's an example of this file. It may be different in the Sakai distribution you are using. You may change it as you are working with Sakai and again change it when you deploy your production Sakai. You can even override the location of the configuration file using the standard log4j procedures. See their documentation for details.

```
# Configures Log4j for Tomcat and Sakai

# use "R" for rolling log separate from catalina.out
#log4j.rootLogger=WARN, R

# use "A" for log in with catalina.out (actually standard output)
log4j.rootLogger=WARN, A

# Configuration for standard output ("catalina.out" in Tomcat).
log4j.appender.A=org.apache.log4j.ConsoleAppender
log4j.appender.A.layout=org.apache.log4j.PatternLayout
log4j.appender.A.layout.ConversionPattern=%p %d %t_%c%n%m%n

# Configuration for a rolling log file ("tomcat.log")
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
log4j.appender.R.DatePattern='.'yyyy-MM-dd
log4j.appender.R.File=/usr/local/tomcat/logs/tomcat.log
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %d %t_%c%n%m%n

# Application logging options
log4j.logger.org.sakaiproject=TRACE
#log4j.logger.org.sakaiproject.component=INFO
#log4j.logger.org.sakaiproject.util=INFO
```

Which Logging Technology To Use?

Using either Commons Logging or log4j in your software has advantages over the Sakai Log API. These produce many more loggers so you have much greater control at runtime about configuring logging in specific areas of your software. The loggers have package like names that form a hierarchy (org, org.sakaiproject, etc). You can control the logging configuration at any level of the hierarchy.

There's not much of a different between using Commons Logging and log4j directly. If you are doing anything fancy in your software with logging, log4j is probably the best choice, as that will present the most possible options. Otherwise it's probably a little safer to use Apache Commons logging, as that is more of an API that supports many different logging technologies (such as log4j, which is a single logging technology).

The Sakai Log API has the same basic programming capabilities as these others, but is limited to using a single logger, so configuration is much more limited.