NGAO Software Architecture
KAON 673: Logging Service

Author	Modified	Notes
Douglas Morrison	6/30/09	NGAO Logging Service

1.0 INTRODUCTION 3

1.1 JAVA LOGGING 3
1.1.1 LOGGER 3
1.1.2 HANDLER 4
1.1.3 FORMATTER 4

2.0 KCSF LOG API 5

2.1 LOG LEVELS 5
2.2 LOG MESSAGE FORMAT 5
2.2.1 STANDARD FORMAT 6
2.2.2 CHARACTER DELIMITED 6
2.3 LOG HANDLERS 6
2.3.1 CONSOLE LOGGER 7
2.3.2 FILE LOGGER 7
2.3.3 DATABASE LOGGER 8
2.4 LOG SERVICE 9

3.0 USE 11

REFERENCES 12

1.0 Introduction

As with any complex software project, systems developed with Keck Common Services Framework (KCSF) will have the need to log a wide range of run-time information from simple trace messages to system critical errors. In order to simplify the logging process a log service will be developed as part of the KCSF infrastructure. The log service will provide developers with the ability to open and write to a multiple log streams, each one with the option of individual customization.

To simplify the development of the log service we will utilize the native Java utility logging package to do the processing and publishing of log information. The KCSF log service will simply wrap the Java classes to hide the majority of the initialization and setup requirements, and provide the standard interface as defined by KCSF.

1.1 Java Logging

The Java logging package follows a simple implementation divided into three concepts: loggers, handlers, and formatters. Loggers are responsible for accepting and processing log messages from the application: they are the main interface to the logging system for clients. Handlers implement the stream functionality of a specific logging mechanism. Common handlers such as the console and file handler are implemented as part of the Java package. Additional handlers can be developed to write to databases or over a network by extending the base *Handler* class. Formatters are responsible for converting a log record into readable text. As with handler's, custom formatters can be developed to suit the user's needs. The Java package also provides a default formatter that implements a simple time stamped event output. Each handler can use its own unique formatter, or the same one can be used for all handlers.

1.1.1 Logger

The Logger class provides a number of methods to log messages and configure a logging system. Messages are logged based on a specified *log level*. The Java utility class defines seven severity levels for log messages,

- FINEST (lowest)
- FINER
- FINE
- CONFIG
- INFO
- WARNING
- SEVERE (highest)

Messages are issued by calling one of the *log* methods with an explicit log level and string message. In addition, helper methods corresponding to each of the log level are also provided to allow clients to implicitly assign a level to a message (e.g. *Logger::warning(…)*). The log level is used to determine if the logger should ignore or process the message. A message will only be processed if its level is greater than or equal to the logger's. The logger level can be set with the *setLevel* method. The logger will pass all log messages that are of a high enough severity level to all of its handlers for publishing.

1.1.2 Handler

The Handler class is the base class for all log publishing objects and is primarily responsible for opening, maintaining, and closing connections to output streams. Log messages are pushed to handlers through the *publish* method. This method accepts a *LogRecord* instance that defines a single log message – including the level, time of creation, and the raw application message. The publish operation is responsible for formatting the log record, and writing it to the output destination.

As with the Logger class, Handlers can be configured to filter messages based on severity levels through the *setLevel* method. This allows a user to restrict the logging of application messages on a handler-to-handler basis.

The Java logging package defines two fully implemented handlers for client use: *ConsoleHandler* and *FileHandler*. The Console Handler is used to write data to an xterm or OS console. The File Handler writes log messages to a file using a rotating file name (as a file reaches a defined maximum file size it is closed and a new file is created appended with an integer suffix.) There is also a base class implementation for a *SocketHandler*, which provide a simple network interface, and a generic *StreamHandler* for stream based logging.

1.1.3 Formatter

The Formatter is a simple class used by Handlers to convert a log record into a human readable string that can be written to an output. Formatters are invoked during the *publish* operation of the handler through the *format* method. This method accepts a raw LogRecord instance and is intended to return a single printable string. The Handler would be responsible for publishing this string to the output.

Only one formatter can be assigned to a handler at a time, but each handler can have its own unique formatting. For consistency it is recommended (and will be enforced by the KCSF) that a single formatter be used for all human readable outputs. For log files that are intended for use by applications or other post processing tools, a character / whitespace delimited format may be required.

The Java logging package provides a simple default formatter for basic logging purposes, as well as an XML formatter that will convert a LogRecord into XML compliant output.

2.0 KCSF Log API

The KCSF Log Service will simplify the process of creating and initializing Java logging by wrapping the classes mentioned previously with KCSF compatible interfaces. Developers will instantiate their desired output handlers and assign them to a Log Service instance. Formatting and handler configuration will automatically be performed by the KCSF framework during Controller initialization.

2.1 Log Levels

Although the Java logging package provides its own severity levels they are rather generic and do not correlate closely enough with actual categories of log messages an application would generate. In addition, with the goal of abstracting away the actual underlying implementation of the logging mechanism, a KCSF specific set of log levels will be defined.

```
public enum LogLevel {
      ALL,          // Log all messages
      TRACE,        // Trace the path of execution
      DEBUG,        // Debug messages
      INFO,         // Information message, normal operation.
      WARN,         // Suspicious operation, possible problem.
      ERROR,        // Recoverable user error has occurred.
      CRITICAL,     // Critical system error, recoverable.
      EMERGENCY,    // Non-recoverable severe failure
      OFF           // Disabled logging
}
```

The *LogLevel* enumeration defines seven severity levels (*TRACE* through *EMERGENCY*) plus an *ALL* and *OFF* value, which will allow through all or none of the log messages, respectively. The KCSF log level will map to the Java logging enumerations as follows:

(KCSF) **Log Level Mappings** (Java)	
TRACE	FINEST
DEBUG	FINER
INFO	FINE
WARN	CONFIG
ERROR	INFO
CRITICAL	WARNING
EMERGENCY	SEVERE
ALL	ALL
OFF	OFF

2.2 Log Message Format

It is common that across projects, or within the same project, multiple log message formats may be used. In some cases this may be required, as the contents of the data or post processing purposes of the data require a specific format. However, in many cases it is simply an oversight or the result of each developer's personal preference.

The Keck Common Services Framework aims to reduce the amount of deviation between log files by encapsulating the selection of the formatters within the framework itself, rather than leaving it directly up to the application programmer. Developer's can offer a restricted set of formats through the implementation of each handler. The programmer would select the appropriate handler for the purpose they need to fulfill. For example, a tab delimited file logging handler can be developed that would produce Excel compatible data.

2.2.1 Standard Format

The following format will be used as the standard for human readable log messages (console, file, etc.)

```
<Time> [<LEVEL>] <Source> - <Message>
```

The *Time* component will be the UT time that the message was logged. Timestamp injection will be performed automatically by the log service, and will have millisecond resolution. The timestamp will have the following format:

```
<Month> <Day>, <Year> <24-HourTime>
```

The *LEVEL* component will be the level of the message – one of the Log Level enumeration values.

Source is the name of the component that generated the log message. This will allow a user to identify the specific software object that produced the message. Source name injection is performed automatically by the log service.

The *Message* component will be the raw message passed to the log service.

A typical log message would look something like:

May 7, 2009 13:14:53.378 [TRACE] ngao.ao.wfs.camera0 – Invalid arguments.

2.2.2 Character Delimited

For the majority of post processing purposes a simple character delimited format would be appropriate. A formatter class will be created that can accept a character as a delimiter between fields. For example a comma delimited message may look something like:

May,7,2009,13:14:53.378,TRACE,ngao.ao.wfs.camera0,Invalid arguments

The formatter may need to be restricted for using certain characters as a delimiter, such as a space.

2.3 Log Handlers

Log handlers implement the actual mechanism used to write log data to a destination. The following details the base class for log handlers.

```
public class ILogHandler {
```

```
      public ILogHandler();

      /**
       * Set the log level.
       */
      public void setLevel(LogLevel level);

      /**
       * Return the current log level.
       */
      public LogLevel getLevel();

      /**
       * Return the handler.
       */
      public java.util.logging.Handler getHandler();


      // The java handler object.
      protected java.util.logging.Handler mHandler;

      // Current log leve.
      protected LogLevel mLevel;
}
```

The base class provides methods to set and get the handler log level, and return a reference to the actual Java handler. Subclasses will be responsible for initializing and assigning a compatible handler during their creation and initialization process.

2.3.1 Console Logger

The Console Logger will provide developers with a simple console logging mechanism. This handler will use the standard formatting only.

```
/**
 * The Console Logger interface.
 */
public interface IConsoleLogger {

}

/**
 * Defines a console logger.
 */
public class ConsoleLogger extends ILogHandler implements IConsoleLogger {

      public ConsoleLogger();

};
```

2.3.2 File Logger

File logging will be the most common logging mechanism used for most projects. The file logger uses the Java *FileHandler* class to manage file streams. The Java handler's rotating file feature will be used to keep a single log file from growing to large. This capability allows a developer to specify a base file name and a maximum acceptable file size. Once the file grows larger than the maximum a

new log file will be created appended with a rotating integer value (0, 1, 2, etc.). The File Logger interface is shown below.

For basic file logging the standard formatter will be used. Log files that require special formatting should be encapsulated in subclasses of the FileLogger implementation.

```
/**
 * The File Logger interface.
 */
public interface IFileLogger {
      public void setFileName(String filename, int maxsize=512000);
}


/**
 * Defines a file logger.
 */
public class FileLogger extends ILogHandler implements IFileLogger {

      public FileLogger();

      /**
       * Set the path and base file name for the log file.
       */
      public void setFileName(String filename, int maxsize=512000);

};
```

2.3.3 Database Logger

KCSF will also provide a logger to save messages in a persistent database. The logger will be responsible for connecting to a specified database and processing messages into appropriate SQL statements. Since databases use their own techniques for formatting data, no formatter is necessary for this logger.

```
public interface IDatabaseLogger {
      public void connect(String utl, String user, String pw);
};

/**
 * Defines a database logger.
 */
public class DatabaseLogger extends ILogHandler implements IDatabaseLogger {

      public DatabaseLogger();

      /**
       * Connect to a database for logging.
       */
      public void connect (String url, String user, String pw);

};
```

2.4 Log Service

The Log Service is the main interface to the logging system as seen by components and their subclasses. The Log Service will implement a basic *log* method that is used to publish a message at a specific severity level. Additional log methods will be provided for each of the severity levels.

```
/**
 * Defines the Logging Service.
 */
public interface ILogService {
	/**
	 * Log a message at the desired level.
	 */
	public void log(LogLevel Level, String Message);

	/**
	 * Produce a TRACE level log message.
	 */
	public void trace(String Message);

	/**
	 * Produce a DEBUG level log message.
	 */
	public void debug(String Message);

	/**
	 * Produce an INFO level log message.
	 */
	public void info(String Message);

	/**
	 * Produce a WARN level log message.
	 */
	public void warn(String Message);

	/**
	 * Produce an ERROR level log message.
	 */
	public void error(String Message);

	/**
	 * Produce a CRITICAL level log message.
	 */
	public void critical(String Message);

	/**
	 * Produce an EMERGENCY level log message.
	 */
	public void emergency(String Message);
}
```

The main implementation of the Log Service will extend the KCSF *AbstractAppServiceTool*, which provides the life-cycle functionality for services. It will also implement an *addHandler* method that will be used by Containers to assign logging objects to the service at startup.

```
/**
 * Defines the Logging Service.
 */
```

```
public class LogService extends AbstractAppServiceTool implements ILogService {

      public LogService();

      /**
       * Add a logging handler.
       */
      public void addHandler(ILogHandler handler);


      ...


      // The java logger object: performs actual logging.
      protected java.util.logging.Logger mLogger;

}
```

As with the KCSF logging implementations discussed previously, the Log Service acts as a wrapper for the Java *Logger* object. This class provides the actual functionality responsible for chaining, filtering, and publishing log messages.

3.0 Use

As with other services, the Log Service is created and initialized during the Container startup process. Each Log Service instance is described by a set of configuration items that define its domain, handlers, and logging properties. The Container will obtain this information and use it to load the class file and create the service and handlers.

The following demonstrates a simple example of how to create a log service with multiple output streams.

```
FileLogger filelog = new FileLogger();
filelog.setFileName(“./logs/ngao.ao.cameras.log”);

ConsoleLogger console = new ConsoleLogger();

DatabaseLogger database = new DatabaseLogger();
database.connect(“127.64.0.123”, “admin”, “none”);

LogService logger = new LogService();
logger.startService(null, “ngao.ao.cameras”);
logger.addHandler(filelog);
logger.addHandler(console);
logger.addHandler(database);

// Sent to file, console and database.
logger.debug(“A debug message”);
```

References

Java Logging Overview:
http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html

Java Logging Pacakage:
http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/package-summary.html