Microlog - a Log4j-based tool for Java ME

Introduction

11 Oct 2009

When we are "logging" codes, we can insert statements into the code and get some useful output information during runtime, such as malfunction code and unexpected errors and behaviors. Examples of logging are trace statements, dumping of structures and the familiar System.out.println or printf debug statements. Although providing scattering and tangling codes - making it difficult to understand, extend and maintain the software - there is no doubt that it can help in many point of views.

Overview

In the J2SE and J2EE platforms, the log4j API offers a hierarchical way to insert logging statements within a Java program. The API offers multiple output formats and multiple levels of logging information, allowing developers to get different kinds of log messages. In the context of the J2ME platform, an interesting Logging API is called MicroLog (created by Johan Karlsson), which is based on the log4j API. For more additional information, see the MicroLog official website.

Recently, I have used the MicroLog API and I had some problems in finding good examples that could help me in using the API. So, in this post, I'll try to show you how nice this API is through the main functionalities provided by the MicroLog API, and how you can use it to do logging. First of all, you can download the latest API release at the source-forge repository. MicroLog is open-source and you can also download the source-code. Become a contributor!

Usage

The use of MicroLog revolves around 3 main classes:

- `public final class Logger`: Logger is responsible for handling the main log operations.
- `public interface Appender`: Appender is responsible for controlling the output of log operations.
- `public interface Formatter`: Formatter is responsible for formatting the output for Appender

With these 3 aforementioned concepts, it is possible to log different kinds of log messages.

Logger Component

The *logger* is the main component. In addition to other operations, the Logger component can be used to set up the Logger Level, such as DEBUG, ERROR or INFO. In the MicroLog API, the following log levels are available:

- static Level DEBUG: The DEBUG Level designates fine-grained informational events that are most useful to debug an application.
- static Level INFO: The INFO level designates informational messages that highlight the progress of the application at coarse-grained level.
- static Level WARN: The WARN level designates potentially harmful situations.
- static Level ERROR: The ERROR level designates error events that might still allow the application to continue running.
- static Level FATAL: The FATAL level designates very severe error events that will presumably lead the application to abort.

A logger will only output messages that are of a level greater than or equal to it. As we can see in Table 1, if you set the Level (or global Level) to WARN, only WARN, ERROR and FATAL will be displayed.

Will Output Messages Of Level

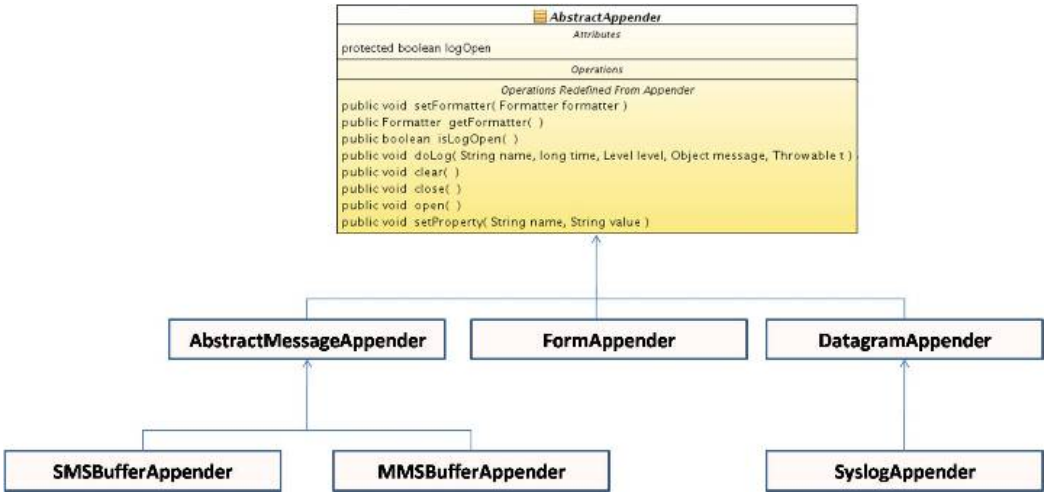
Logger Level	DEBUG	INFO	WARN	ERROR	FATAL
DEBUG					
INFO					
WARN					
ERROR					
FATAL					

To initiate your loggin process, first you can create a new Logger instance (as discussed ealier). There are three ways to do that, such as:

```
Logger logger = LoggerFactory.getLogger(); Creates a logger without a name
Logger logger = LoggerFactory.getLogger(String loggerName); Creates a logger passing a logger name (loggerName variable)
Logger logger = LoggerFactory.getLogger(Class class); Creates a logger with a class reference, such as your MIDlet class.
```

Appender Component

The process of logging requires that you define the messages output interface, such as to a file, to a console or to a bluetooth connection. Like Log4j, MicroLog also defines many kinds of appenders. In the context of the J2ME platform, there are many appenders that can be used in the MIDP to send your log messages, such as RecordStoreAppender, where you can store your log messages in the RecordStore, BluetoothSerialAppender, where you can send your log messages to a bluetooth connection and FormAppender, where you can show the messages into a Form (LCDUI) interface. The following Figure ilustrates a summarized UML that shows how they are organized.



The *AbstractAppender* defines an interface with a set of methods where all other appenders should redefine them in their own class. For instance, consider a *FormAppender*. It uses the *doLog* method to send (show) log messages to a *Form* reference (*form.append()*), *clear()* for cleaning up all messages (*form.deleteAll()*), and *open()* to create a new Form (default) instance.

In this section, I'll show you how to send log messages to a *Form* (LCDUI) interface and to a bluetooth connection. The former is shown in our first example. After creating a Logger reference (which has "Form Logger" as its name), we created a new *FormAppender* instance, passing our *Form* reference (which will display the log messages). *log.addAppender(appender)* can be used to associate our new appender reference. It is important to point out that you can create as many appenders as you wish, just calling *addAppender()*. As a result, you can send log messages, at the same time, to a File (through *FileConnectionAppender*) and to a datagram connection.

```
public class FormExample extends MIDlet {
    public FormExample() {
        this.d       = Display.getDisplay(this); //creates display instance
        this.f       = new Form("Loggin"); //creates a form

        this.log     = LoggerFactory.getLogger("Form Logger"); //creates a Logger
        this.appender = new FormAppender(this.f); //creates an appender to add log outputs

        this.log.addAppender(this.appender); //adds the output log to a form
        this.log.debug("Constructed object!"); //logs a message
    }

    protected void destroyApp(boolean arg0) throws MIDletStateChangeException {}
    protected void pauseApp() {}

    protected void startApp() throws MIDletStateChangeException {
        this.d.setCurrent(this.f);
        this.log.debug("Starting midlet..."); //logs a message
    }
}
```

And the result is:

Our next example shows how to send log messages to a bluetooth connection. The *BluetoothSerialAppender* can be used to this goal, which uses the btspp protocol and can be used in two different modes: (1) The implementation tries to find the Bluetooth logger server through bluetooth lookup services, or (2) by specifying the exact url of the server, which is usefull for devices that fails to lookup the server. Microlog also has two types of server: (1) Datagram server and a (2) bluetooth server. You can download both here.

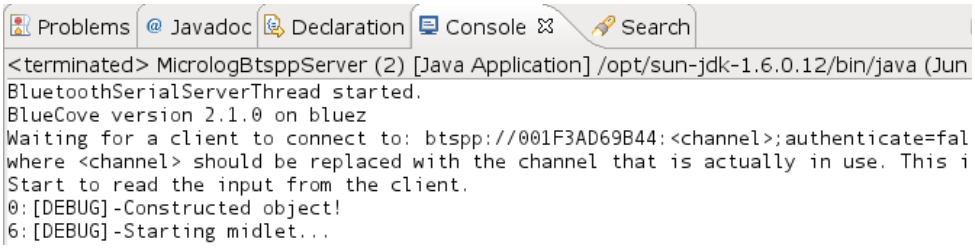
The following code demonstrates how to use *BluetoothSerialAppender*:

```
public class BluetoothExample extends MIDlet {
```

```
    public BluetoothExample() {
        this.log = LoggerFactory.getLogger(BluetoothExample.class); //creates a logger instance
        this.log.setLogLevel(Level.DEBUG); //sets the log level
        //creates a bluetooth appender
        this.appender = new BluetoothSerialAppender("btspp://001F3AD69B44:1;authenticate=false;encrypt=false;master=false");
        this.log.addAppender(appender); //adds the appender to the log reference
        this.log.debug("Constructed object!"); //logs a message
    }

    protected void destroyApp(boolean arg0) throws MIDletStateChangeException {}
    protected void pauseApp() {}

    protected void startApp() throws MIDletStateChangeException {
        this.log.debug("Starting midlet...");
    }

}
```

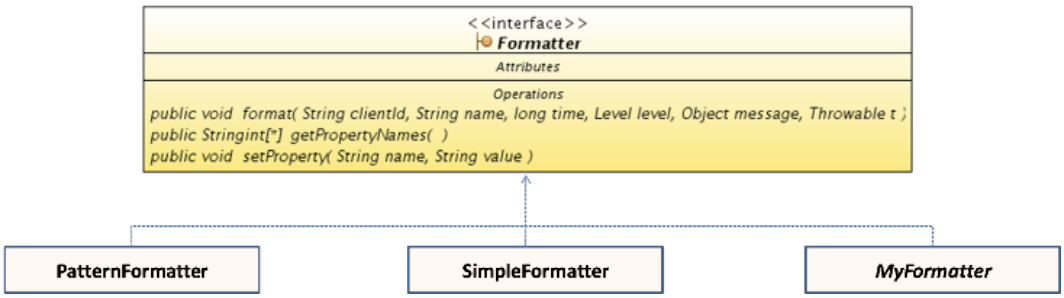
In our example, we passed the exactly bluetooth server URL: btspp://001F3AD69B44:<channel>;authenticate=false;encrypt=false;master=false, where you can replace "<channel>" to a low number 1 or 2 (depending on which is being used). After the double slashes, 001F3AD69B44 represents the bluetooth logger server. The result of this example is shown in the next Figure, where I used my desktop as the bluetooth server.

WARNING: Be careful when using bluetooth-based logging. If your application already uses bluetooth connections, bluetooth buffer can get full if many data is sent and received!



Formatter Component

To format a message, an Appender must have an associated Formatter object. MicroLog has two different formatters: (1) The SimpleFormatter is the default and the simplest formatter, and (2) the PatternFormatter offers more flexibility for choosing which information will appear in the log message. If you are not satisfied with them, you can also create your own formatter class (discussed later) and define your own message format. Formatters implement the same interface (Formatter), which can be seen in the following Figure. The format() method is where the formatting of the message really occurs, where we have the logger name, the logger level, a related message and a throwable object.



The following steps show how to associate a formatter to an appender reference (in this context, the log messages will be sent to the console).

```
Logger log = LoggerFactory.getLogger("Logger With SimpleFormatter");
ConsoleAppender appender = new ConsoleAppender(); //creates a Console appender
Formatter formatter = new SimpleFormatter(); //creates a simpleFormatter
appender.setFormatter(formatter); //adds a formatter to a specific appender
log.addAppender(appender); //adds the appender to the logger
log.debug("Constructed object!"); //logs a message
```

The SimpleFormatter would print:

```
 0:[DEBUG]-Constructed object!
```

As we can see, the SimpleFormatter is very simple. However, the PatternFormatter offers a more sofisticated way to format log messages. It works by defining your own formatting pattern, where you can choose which information will be in the message. For instance, consider our next example:

```
Logger log = LoggerFactory.getLogger();
PatternFormatter formatter = new PatternFormatter(); //Creates a PatternFormatter
formatter.setPattern("%t %d [%P] %m %T"); //specifies which data will be appended to the message
Appender appender = new ConsoleAppender();
appender.setFormatter(formatter);
log.addAppender(this.appender);
```

The message ouput would be:

```
 Thread-0 12:54:18,815 [DEBUG] Starting app...
```

Note that the message format was constructed based on some "special characters" using *formatter.setPattern()*. The following list shows which types of characters you can use:

- %c - prints the name of the Logger
- %d - prints the date (absolute time)
- %m - prints the logged message
- %P -prints the priority, i.e. Level of the message.
- %r - prints the relative time of the logging. (The first logging is done at time 0.)
- %t - prints the thread name.
- %T - prints the Throwable object.
- %% - prints the '%' sign.

If you are not satiesfied with these two aforementioned formatters, you can also create your own one. For instance, let's say that we want to format a log message as follows:

```
 $ sequenceNumber $ Log Level $ Message $
```

To create our own formatter, let's just implement the *Formatter* interface and define the message in the *format* method. The example defines an int primitive type (to show the sequence number) and a delimeter ($) to separete each data type (sequence number, log level and message).

```
public class MyFormatter implements Formatter {

    private static final String DELIMETER = " $ ";
    private int sequence;
    private StringBuffer buffer;

    public MyFormatter() {
        this.sequence = 0; //the sequence of messages
        this.buffer   = new StringBuffer(); //message to be shown
    }

    public String format(String clientID, String name, long time, Level level,
            Object message, Throwable t) {

        this.buffer.delete(0, buffer.length()); //delete previously logged message

        this.buffer.append(DELIMETER); //appends delimeter
        this.buffer.append(this.sequence++); //increments sequence message number
        this.buffer.append(DELIMETER); //appends delimeter
        this.buffer.append(level); //appends the log level
        this.buffer.append(DELIMETER); //appends delimeter
        this.buffer.append(message); //appends log messages
        this.buffer.append(DELIMETER); //appends delimeter

        return buffer.toString(); //creates the entire log message
    }
}
```

The result is:

```
 $ 0 $ DEBUG $ Constructed object! $
 $ 1 $ DEBUG $ Starting middlet... $
```

Using a Configuration File

MicroLog also offers mechanisms to insert your log configurations into a configuration file, instead of setting them within the software. The advantage of using external files is that changes in the log configurations do not imply to recompile the software. However, due to io instructions, the process can be slower.

The following listing shows a configuration file that sets the log level to WARN, uses two different appenders (console and file) and one formatter (*PatternFormatter*). The configuration file is called microlog.properties and can be saved into the */res* project folder.

```
 microlog.level=WARN
 microlog.appender=net.sf.microlog.core.appender.ConsoleAppender;net.sf.microlog.midp.appender.FileConnectionAppender
 microlog.formatter=net.sf.microlog.common.format.PatternFormatter
 microlog.formatter.PatternFormatter.name=MyFormatterName
 microlog.formatter.PatternFormatter.pattern=%c %d [%P] %m %T
 # End of file.
```

The PropertyConfigurator class is responsible for loading the configuration file.

```
public class PropertiesExample extends MIDlet {

    private Logger log;

    public PropertiesExample() {
        this.log = LoggerFactory.getLogger();
        PropertyConfigurator.configure("/microlog.properties"); //loads the configuration file
    }

    protected void destroyApp(boolean arg0) throws MIDletStateChangeException {}
    protected void pauseApp() {}

    protected void startApp() throws MIDletStateChangeException {
        this.log.info("info");
        this.log.error("Constructed object! erro");
        this.log.debug("debug");
        this.log.fatal("afta");
        this.log.warn("warn");
    }
}
```

The output can be one as shown below. First, note that only ERROR, FATAL and WARN messages were sent to the ouput, because we set our log level to WARN. Second, the log messages were also sent to a file, created at *///root1/microlog.txt*.

```
Loading properties from /microlog.properties
Added appender net.sf.microlog.core.appender.ConsoleAppender@1cb37664
Added appender net.sf.microlog.midp.appender.FileConnectionAppender@f828ed68
Using formatter class net.sf.microlog.common.format.PatternFormatter
The created file is file:///root1/microlog.txt
20:34:43,151 [ERROR] Constructed object! error
20:34:44,119 [FATAL] afta
20:34:44,809 [WARN] warn
```

Finally, MicroLog framework seems to be a good contribution for the JavaME platform due to its simplicity (based on the Log4j framework) and the lack of good tools to debug JavaME applications when they are running in a real mobile device.