LoggingFilter discussion

Introduction

The **loggingFilter** class has been created to provide logs on events.

Discussion

The current implementation is overly complex. We have had a discussion about how to implement it better. Some suggestion have been pushed :

- remove the Map from the logger, and fix the code to behave as the initial implementation (in term of features)

- use only one Logger,

- or simply remove the filter completely, as we can always set the log using the proper configuration

We still have to select what is the best option.

Implementation proposal

Here is some code I have written to replace the current implementation. It has the very same features, but with less code, more java documentation and no synchronization.

It also use only one logger for all the eventType, which should be set to TRACE when configuring the log system, otherwise the eventType Loglevel will potentially be overridden by the log system level.

```
package org.apache.mina.filter.logging;

import org.apache.mina.core.filterchain.IoFilter;
import org.apache.mina.core.filterchain.IoFilterAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoEventType;
import org.apache.mina.core.session.IoSession;
import org.apache.mina.core.write.WriteRequest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Logs all MINA protocol events.  Each event can be
 * tuned to use a different level based on the user's specific requirements.  Methods
 * are in place that allow the user to use either the get or set method for each event
 * and pass in the {@link IoEventType} and the {@link LogLevel}.
 *
 * By default, all events are logged to the {@link LogLevel#INFO} level except
 * {@link IoFilterAdapter#exceptionCaught(IoFilter.NextFilter, IoSession, Throwable)},
 * which is logged to {@link LogLevel#WARN}.
 *
 * @author The Apache MINA Project (dev@mina.apache.org)
 * @version $Rev: 671827 $, $Date: 2008-06-26 10:49:48 +0200 (Thu, 26 Jun 2008) $
 */
public class LoggingFilter extends IoFilterAdapter {
        /** The logger name */
    private final String name;

    /** The logger */
    private final Logger logger;

    /** The log level for the exceptionCaught event. Default to WARN. */
    private LogLevel exceptionCaughtLevel = LogLevel.WARN;

    /** The log level for the messageSent event. Default to INFO. */
    private LogLevel messageSentLevel = LogLevel.INFO;

    /** The log level for the messageReceived event. Default to INFO. */
    private LogLevel messageReceivedLevel = LogLevel.INFO;

    /** The log level for the sessionCreated event. Default to INFO. */
    private LogLevel sessionCreatedLevel = LogLevel.INFO;
```

```
/** The log level for the sessionOpened event. Default to INFO. */
private LogLevel sessionOpenedLevel = LogLevel.INFO;

/** The log level for the sessionIdle event. Default to INFO. */
private LogLevel sessionIdleLevel = LogLevel.INFO;

/** The log level for the sessionClosed event. Default to INFO. */
private LogLevel sessionClosedLevel = LogLevel.INFO;

/**
 * Default Constructor.
 */
public LoggingFilter() {
    this(LoggingFilter.class.getName());
}

/**
 * Create a new LoggingFilter using a class name
 *
 * @param clazz the cass which name will be used to create the logger
 */
public LoggingFilter(Class<?> clazz) {
    this(clazz.getName());
}

/**
 * Create a new LoggingFilter using a name
 *
 * @param name the name used to create the logger. If null, will default to "LoggingFilter"
 */
public LoggingFilter(String name) {
    if (name == null) {
        this.name = LoggingFilter.class.getName();
    } else {
            this.name = name;
    }

    logger = LoggerFactory.getLogger(name);
}

/**
 * @return The logger's name
 */
public String getName() {
    return name;
}

/**
 * Log if the logger and the current event log level are compatible. We log
 * a message and an exception.
 *
 * @param eventLevel the event log level as requested by the user
 * @param message the message to log
 * @param cause the exception cause to log
 */
private void log(LogLevel eventLevel, String message, Throwable cause) {
        if (eventLevel == LogLevel.TRACE) {
                logger.trace(message, cause);
        } else if (eventLevel.getLevel() > LogLevel.INFO.getLevel()) {
                logger.info(message, cause);
        } else if (eventLevel.getLevel() > LogLevel.WARN.getLevel()) {
                logger.warn(message, cause);
        } else if (eventLevel.getLevel() > LogLevel.ERROR.getLevel()) {
                logger.error(message, cause);
        }
}

/**
 * Log if the logger and the current event log level are compatible. We log
 * a formated message and its parameters.
```

```
 *
 * @param eventLevel the event log level as requested by the user
 * @param message the formated message to log
 * @param param the parameter injected into the message
 */
private void log(LogLevel eventLevel, String message, Object param) {
        if (eventLevel == LogLevel.TRACE) {
                logger.trace(message, param);
        } else if (eventLevel.getLevel() > LogLevel.INFO.getLevel()) {
                logger.info(message, param);
        } else if (eventLevel.getLevel() > LogLevel.WARN.getLevel()) {
                logger.warn(message, param);
        } else if (eventLevel.getLevel() > LogLevel.ERROR.getLevel()) {
                logger.error(message, param);
        }
}

/**
 * Log if the logger and the current event log level are compatible. We log
 * a simple message.
 *
 * @param eventLevel the event log level as requested by the user
 * @param message the message to log
 */
private void log(LogLevel eventLevel, String message) {
        if (eventLevel == LogLevel.TRACE) {
                logger.trace(message);
        } else if (eventLevel.getLevel() > LogLevel.INFO.getLevel()) {
                logger.info(message);
        } else if (eventLevel.getLevel() > LogLevel.WARN.getLevel()) {
                logger.warn(message);
        } else if (eventLevel.getLevel() > LogLevel.ERROR.getLevel()) {
                logger.error(message);
        }
}

@Override
public void exceptionCaught(NextFilter nextFilter, IoSession session,
        Throwable cause) throws Exception {
        log(exceptionCaughtLevel, "EXCEPTION :", cause);
    nextFilter.exceptionCaught(session, cause);
}

@Override
public void messageReceived(NextFilter nextFilter, IoSession session,
        Object message) throws Exception {
        log(messageReceivedLevel, "RECEIVED: {}", message );
        nextFilter.messageReceived(session, message);
}

@Override
public void messageSent(NextFilter nextFilter, IoSession session,
        WriteRequest writeRequest) throws Exception {
        log(messageSentLevel, "SENT: {}", writeRequest.getMessage() );
    nextFilter.messageSent(session, writeRequest);
}

@Override
public void sessionCreated(NextFilter nextFilter, IoSession session)
                throws Exception {
        log(sessionCreatedLevel, "CREATED");
    nextFilter.sessionCreated(session);
}

@Override
public void sessionOpened(NextFilter nextFilter, IoSession session)
throws Exception {
        log(sessionOpenedLevel, "OPENED");
    nextFilter.sessionOpened(session);
}
```

```
@Override
public void sessionIdle(NextFilter nextFilter, IoSession session,
        IdleStatus status) throws Exception {
        log(sessionIdleLevel, "IDLE");
    nextFilter.sessionIdle(session, status);
}

@Override
public void sessionClosed(NextFilter nextFilter, IoSession session) throws Exception {
        log(sessionClosedLevel, "CLOSED");
    nextFilter.sessionClosed(session);
}

/**
 * Set the LogLevel for the ExceptionCaught event.
 *
 * @param level The LogLevel to set
 */
public void setExceptionCaughtLoglevel(LogLevel level) {
        exceptionCaughtLevel = level;
}

/**
 * Get the LogLevel for the ExceptionCaught event.
 *
 * @return The LogLevel for the ExceptionCaught eventType
 */
public LogLevel getExceptionCaughtLoglevel() {
        return exceptionCaughtLevel;
}

/**
 * Set the LogLevel for the MessageReceived event.
 *
 * @param level The LogLevel to set
 */
public void setMessageReceivedLoglevel(LogLevel level) {
        messageReceivedLevel = level;
}

/**
 * Get the LogLevel for the MessageReceived event.
 *
 * @return The LogLevel for the MessageReceived eventType
 */
public LogLevel getMessageReceivedLoglevel() {
        return messageReceivedLevel;
}

/**
 * Set the LogLevel for the MessageSent event.
 *
 * @param level The LogLevel to set
 */
public void setMessageSentLoglevel(LogLevel level) {
        messageSentLevel = level;
}

/**
 * Get the LogLevel for the MessageSent event.
 *
 * @return The LogLevel for the MessageSent eventType
 */
public LogLevel getMessageSentLoglevel() {
        return messageSentLevel;
}

/**
 * Set the LogLevel for the SessionCreated event.
 *
 * @param level The LogLevel to set
```

```
     */
    public void setSessionCreatedLoglevel(LogLevel level) {
            sessionCreatedLevel = level;
    }

    /**
     * Get the LogLevel for the SessionCreated event.
     *
     * @return The LogLevel for the SessionCreated eventType
     */
    public LogLevel getSessionCreatedLoglevel() {
            return sessionCreatedLevel;
    }

    /**
     * Set the LogLevel for the SessionOpened event.
     *
     * @param level The LogLevel to set
     */
    public void setSessionOpenedLoglevel(LogLevel level) {
            sessionOpenedLevel = level;
    }

    /**
     * Get the LogLevel for the SessionOpened event.
     *
     * @return The LogLevel for the SessionOpened eventType
     */
    public LogLevel getSessionOpenedLoglevel() {
            return sessionOpenedLevel;
    }

    /**
     * Set the LogLevel for the SessionIdle event.
     *
     * @param level The LogLevel to set
     */
    public void setSessionIdleLoglevel(LogLevel level) {
            sessionIdleLevel = level;
    }

    /**
     * Get the LogLevel for the SessionIdle event.
     *
     * @return The LogLevel for the SessionIdle eventType
     */
    public LogLevel getSessionIdleLoglevel() {
            return sessionIdleLevel;
    }

    /**
     * Set the LogLevel for the SessionClosed event.
     *
     * @param level The LogLevel to set
     */
    public void setSessionClosedLoglevel(LogLevel level) {
            sessionClosedLevel = level;
    }

    /**
     * Get the LogLevel for the SessionClosed event.
     *
     * @return The LogLevel for the SessionClosed eventType
     */
    public LogLevel getSessionClosedLoglevel() {
            return sessionClosedLevel;
    }
}
```

The **LogLevel** class is :

```
package org.apache.mina.filter.logging;

/**
 * Defines a logging level.
 *
 * @author The Apache MINA Project (dev@mina.apache.org)
 * @version $Rev: 644681 $, $Date: 2008-04-04 13:10:59 +0200 (Fri, 04 Apr 2008) $
 *
 * @see LoggingFilter
 */
public enum LogLevel {

    /**
     * {@link LogLevel} which logs messages on the TRACE level.
     */
        TRACE(5),

    /**
     * {@link LogLevel} which logs messages on the DEBUG level.
     */
        DEBUG(4),

    /**
     * {@link LogLevel} which logs messages on the INFO level.
     */
        INFO(3),

    /**
     * {@link LogLevel} which logs messages on the WARN level.
     */
        WARN(2),

    /**
     * {@link LogLevel} which logs messages on the ERROR level.
     */
        ERROR(1),

    /**
     * {@link LogLevel} which will not log any information
     */
        NONE(0);

        /** The internal numeric value associated with the log level */
        private int level;

        /**
         * Create a new instance of a LogLevel.
         *
         * @param level The log level
         */
        private LogLevel(int level) {
                this.level = level;
        }


        /**
         * @return The numeric value associated with the log level
         */
        public int getLevel() {
                return level;
        }
}
```