Logging with sf-loggingservices

9 May 2008

1 Introduction

The `sf-loggingservices` components are a set of extensions to the logging framework used by the SmartFrog core. These are intended integrate SmartFrog logging with the Apache loggers, Log4J and Commons-Logging, [log4j,jcl]. There are some very good reasons for doing this

1. Many third party libraries use Log4J for logging, or commons-logging to bind to a system logger. By using the same logging framework for SmartFrog, all log messages are routed to the same place
2. Log4J comes with some very powerful log tools, such as the rolling HTML logger, or the Chainsaw distributed log viewer.

SmartFrog can not only log through Log4J, it can help set up the rest of the system to log correctly. For example, it can deploy a Jetty or Tomcat instance to serve up the generated log files, making them remotely visible. This is invaluable for deployment to remote systems.

At the time of writing (February 2007), the commons-logging integration is not complete, so it is not discussed. Readers are directed to the source for more details.

2 Log4J Wrapper

This wrapper redirects all logging statements from the core logging framework to the Apache Log4J logging framework,. (http://logging.apache.org/log4j/).

This component has been built and tested against Log4J version 1.2.13.

2.1 Component Description

The component is included with the sf-loggingservices module on the classpath:

```
#include "/org/smartfrog/services/logging/log4j/logtolog4jimpl.sf"
```

The file declares the logger `LogToLog4JImpl`:

```
LogToLog4JImpl extends {
   setIniLog4JLoggerLevel false;
   ignoreSetLogLevel false;
   loggerClass "org.smartfrog.services.logging.log4j.LogToLog4JImpl";
}
```

The component supports the following attributes:

Attribute Name	*Description*	*Optional/Mandatory*
`configuratorFile`	URI to a properties or XML Log4J configuration file.	Optional
`resource`	Name of a resource (on the classpath) to load for the configuration	Optional
`setIniLog4JLoggerLevel`	Boolean to determine if to set initial Log4J level. Default false.	Optional
`ignoreSetLogLevel`	Boolean to ignore `setLogLevel` method. Default false.	Optional
`logLevel`	Value for the log level	Optional

The first two configuration attributes select the Log4J configuration file. The component will also use the System property `log4j.configuration` if it is set and neither the `configuratorFile` nor the `resource` attribute is set. This is for consistency with normal Log4J operation.

2.2 Usage

To use the Log4J components, you must have on the classpath of the application (or SmartFrog itself)

- `sf-loggingservices.jar`
- `log4j.jar`, preferably `log4j-1.2.13.jar` or later
- `commons-logging.jar`. This is is optional, but strongly recommended as it may be required in future uses of the loggingservices module. Use version 1.1.1

Logging is one of the basic services that the framework needs and it is therefore initialized early on during the daemon's startup phase.

To set the initial configuration for the core logging framework define the property

```
org.smartfrog.sfcore.logging.LogImpl.localLoggerClass
```

-in this case with the value

```
org.smartfrog.services.logging.log4j.LogToLog4JImpl
```

This can be done by editing the configuration file `default.ini`.

Example 1: default.ini configuration

In the daemon's `default.ini` configuration file:

Select the Log4j logger:

```
org.smartfrog.sfcore.logging.LogImpl.localLoggerClass=
  org.smartfrog.services.logging.log4j.LogToLog4JImpl
```

Configure this with an XML configuration file:

```
org.smartfrog.sfcore.logging.logger.LogToLog4JImpl.configuratorFile=d:\\log4j.xml
```

This will make Log4J the logger for the entire daemon, using the Log4J XML file specified in the output file.

It is also possible to add more than one logger to the log framework and modify their configuration at runtime.

Example 2: Multiple loggers in default.ini

SmartFrog can actually log to multiple loggers simultaneously, so Log4J can be used alongside any other loggers, by listing them all in the `org.smartfrog.sfcore.logging.LogImpl.loggerClass` property as a list using the `[| |]` list construct.

```
org.smartfrog.sfcore.logging.LogImpl.loggerClass=[|"org.smartfrog.sfcore.\
  .logging.LogToStreamsImpl","org.smartfrog.sfcore.logging.LogToFileImpl",\
  "org.smartfrog.services.logging.log4j.LogToLog4JImpl"|]
```

Example 3: configuration inside a deployment descriptor to log HTML pages to a web site

You can also set up the logger from within SmartFrog. This is done by binding a component to the log. Note that by doing so, you reset the entire log4J configuration.

To set up an application that logs to Log4J, we have to set up a specific log binding for the component hierarchy that we wish to log to:

```
  log extends SFLog {
    //log against our parent
    logFrom LAZY PARENT;
    //now log4j kicks in
    logTo extends LogToLog4JImpl {
      resource "log4j.properties";
    }
```

```
  }
```

The result is that everything is logged to the according to the specification in the resource log4j.properties on the classpath. This file is set up to log to the console:

```
log4j.rootCategory=INFO, CONSOLE


log4j.logger.org.smartfrog=INFO
log4j.logger.org.smartfrog.sfcore.languages.sf=DEBUG
log4j.logger.org.smartfrog.services.xml=DEBUG
log4j.logger.org.smartfrog.services.database=DEBUG

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=DEBUG
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%-4r %-5p %c %x - %m%n
```

When used, this configuration will log to the console

```
0    INFO  HOST localhost:rootProcess:TableManipulationTest:action:log  - logger:
 "HOST localhost:rootProcess:TableManipulationTest:action:log" (INFO)
  with configuration file:/home/user/Projects/SmartFrog/Forge/core/components/database/
     build/test/classes/log4j.properties
```

By altering the `CONSOLE.layout` properties, the structure of this message can be altered.

Example 4: Logging to rolling HTML pages served up on a deployed web server

A variation on the previous example is to modify the Log4J properties file to log out HTML pages:

```
# This is the log4j settings for the deployment with rolling html output

log4j.rootCategory=INFO, WWW

log4j.logger.org.smartfrog=INFO
log4j.logger.org.smartfrog.sfcore.languages.sf=DEBUG
log4j.logger.org.smartfrog.services.xml=DEBUG
log4j.logger.org.smartfrog.services.database=DEBUG

log4j.appender.WWW=org.apache.log4j.DailyRollingFileAppender
log4j.appender.WWW.DatePattern='.'yyyy-MM-dd-HH'.html'
log4j.appender.WWW.File=/home/user/public_html/logs/server-log.html
log4j.appender.WWW.layout=org.apache.log4j.HTMLLayout
log4j.appender.WWW.layout.LocationInfo=true
```

This sets up the log to generate HTML files in the `/home/user/public_html/logs/` directory, rolling the files every day. Unfortunately, Log4J does not create directories on demand -the output directory must be there before logging starts. This can be created with a `MkDir` component, that creates the target directory inline. We also need to serve up the the HTML output, for which the Jetty component in the sf-jetty module comes in to play: it will create a servlet context that serves up the log directory under the /logs context of the web server:

```
#include "/org/smartfrog/services/logging/log4j/logtolog4jimpl.sf"
#include "/org/smartfrog/sfcore/logging/components.sf"
#include "/org/smartfrog/services/jetty/jetty-complete.sf"


HttpLogServer extends Compound {


   homedir "/home/user/public_html";

   logDirectory extends Mkdir {
    dir "logs";
    parentDir homedir;
   }


  log extends SFLog {
    //log against our parent
    logFrom LAZY PARENT;
    //now log4j kicks in
    logTo extends LogToLog4JImpl {
```

```
      resource "log4j.properties";

    }
  }

   jettyServer extends CoreJettyServer {
      jettyhome PARENT:homedir;
      port 8080;
      server "logserver";
    }

  /**
   * Serving up of logs
   */
  logs extends ServletContext {

      resourceBase LAZY logDirectory:absolutePath;
      contextPath "/logs";
      server LAZY jettyServer;

   }

  logServlets extends Compound {
    context LAZY logs;
    directory extends SrcDefaultServlet {
      pathSpec "/*";
      context LAZY PARENT:context;
    }

  }
    //mime type of html pages
    htmlPages extends MimeType {
      extension "html";
      type "text/html";
      context LAZY logs;
     }

}
```

The result is that a single deployment will use Log4J for logging, *and host the log where it is publicly visible*.

There are two security risks with this. One is that it may expose private data. The other is that if anything gets logged that is not correctly escaped, the entire domain is potentially exposed to Cross Site Scripting (XSS) attacks, in which malicious JavaScript code can acquire login cookies and access authenticated services in the same domain. Production environments may wish to consider securing access, or expose the web site on a port that is not allowed through the firewall.

3 How Log4J logs and levels are set up

You can set the initial log level in the `logLevel` attribute, which is only used if the `setIniLog4JLoggerLevel` is set to true;

```
#include "/org/smartfrog/services/logging/log4j/logtolog4jimpl.sf"
#include "/org/smartfrog/sfcore/logging/logimpl.sf"
#include "/org/smartfrog/sfcore/logging/components.sf"

sfConfig extends Compound {
  log extends SFLog {
    //log against our parent
    logFrom LAZY PARENT;
    //now log4j kicks in
    logTo extends LogToLog4JImpl {

      resource "log4j.properties";
      setIniLog4JLoggerLevel true;
    }
    logLevel LOG_LEVEL_DEBUG;
  }
}
```

This forces the logger to log at the `LOG_LEVEL_DEBUG` level; a constant that comes from the `logimpl.sf` file.

When Log4J is on the classpath, any libraries written against the Log4J APIs automatically bind to it and output over the same loggers. They will use the resource `/log4j.properties` or `/log4j.xml`, or if set, the URL/resource specified in the system property `log4j.configuration`.

Furthermore, any code that uses commons-logging automatically gets switched to Log4J in preference to the Sun logging API. All such classes normally ask for a logger based on the class name of the component

```
Log=LogFactory.create(this.getClass())
```

These logger configurations are constant for a single JVM; the only way to let these classes log at different levels in different applications is to deploy the different applications in different JVMs, with different Log4J configuration files.

In contrast to being classname-driven, deployed components picks up their logger names from the component hierarchy. You can see the component name on the initial deployment, along with the fact that the logger is now logging at the DEBUG level.

```
0    INFO  HOST localhost:rootProcess:TableManipulationTest:action:log  - logger:
 "HOST localhost:rootProcess:TableManipulationTest:action:log" (DEBUG)
with configuration file:/home/user/Projects/SmartFrog/Forge/core/components/database
 /build/test/classes/log4j.properties
```

Here the name of the logger is `"HOST localhost:rootProcess:TableManipulationTest:action:log"`. This is not something easily used in a properties file, and varies with every deployment. You can specify a new name with the `logName` attribute of the `SFLog` component:

```
#include "/org/smartfrog/services/logging/log4j/logtolog4jimpl.sf"

sfConfig extends Compound {

  log extends SFLog {
    logName "org.smartfrog.org.smartfrog.services.database";
    logFrom LAZY PARENT:mysqld;
    logTo extends LogToLog4JImpl {
      resource "log4j.properties";
    }
  }

  mysqld extends Mysqld {
    datadir LAZY PARENT:datadir:absolutePath;
    basedir datadir;
  }

}
```

This will use the **`"org.smartfrog.org.smartfrog.services.database"`** log, which in the log4j.properties files of our examples has been set to log at the DEBUG level.

The key point to note here is that the name is set in a specific `SFLog` log binding, not in the `LogToLog4JImpl` declaration. For effective Log4J logging, the log name should be that of a Log4J package hierarchy, with the result that the output appears something like the following:

```
398141 DEBUG org.smartfrog.org.smartfrog.services.database  - ConfigURL (from
log4j.configuration sys property): null
398141 [RMI TCP Connection(5)-127.0.0.1]
  DEBUG org.smartfrog.org.smartfrog.services.database
   - ConfigURL (from log4j.configuration sys property): null
LogToLog4JImpl: Using Log4J.PropertyConfigurator with URL
  file:/home/user/Projects/SmartFrog/Forge/core/components/database
    /build/test/classes/log4j.properties
398145 INFO  org.smartfrog.org.smartfrog.services.database
  - LogToLog4JImpl: Using Log4J.PropertyConfigurator with URL
   file:/home/user/Projects/SmartFrog/Forge/core/components/database
    /build/test/classes/log4j.properties
398145 DEBUG org.smartfrog.org.smartfrog.services.database
  - class org.smartfrog.services.logging.log4j.LogToLog4JImpl
   'org.smartfrog.org.smartfrog.services.database' using ComponentDescription:
setIniLog4JLoggerLevel false;
ignoreSetLogLevel false;
loggerClass "org.smartfrog.services.logging.log4j.LogToLog4JImpl";
```

This trace actually shows the startup process of the logger. What happens when the logger is deployed that first it binds Log4J to the default configuration (here the `log4.configuration` system property), then the configuration description of the `sfConfig:log:logTo` component is read, which provides the `resource` attribute that sets up the final log configuration. This is then loaded and we switch over to the application-specific log.

> **Important**: Loading a new Log4J configuration can reset *all* existing configurations. Although a single Log4J configuration can define many different logging options for different loggers, you cannot load different Log4J properties or XML files for different applications. Whichever component loads a configuration last, wins.

The active Log4J configuration is not lost when an application with a Log4J configuration is terminated. That is, it remains active for the duration of the specific JVM, or until a new configuration is loaded.

3.1 Detailed Debugging of Log4J

It is a little known fact that Log4J can itself be debugged using system properties

Property	*Meaning*
`log4j.debug`	If set print log4j-internal debug statements to `System.out`.

This property can be set to any non-null value to trace exactly what goes on inside a JVM.

4 Log4J best practises

- If you are planning to use Log4J in production, start using it on development systems to gain experience in configuring it, and in reading the results.
- Library code that is designed to be used outside SmartFrog should either use Log4J or commons logging as a logging API. The Advantage of commons-logging is that it permits multiple back ends, including the Sun logger, and, in future, the SmartFrog logging infrastructure. The disadvantage is that it is another layer of indirection, and can suffer from classpath problems if different loggers are loaded in different classloaders [jclts].
- Consider creating a special log for logging security or other log details, so that Log4J can be set up to route these messages differently.
- Don't log in such a way that the log methods themselves can raise null pointer exceptions or similar. Logging should not crash the application.
- Never ship a JAR with a resource called `/log4j.properties`. If logging does not behave, check third party JAR files (such as `Axis.jar`) for such a file, and delete it.
- Never ship with a resource called `/log4.properties` in a SmartFrog JAR that is intended for reuse, such as any of the redistributable SmartFrog JARs.
- If you are logging to a directory, create this directory in the deployment descriptor.
- Do not rely on clocks being synchronized across machines, when trying to post-mortem activities from the logs.

5 Commons-Logging Integration

There is a bridge from Apache Commons-Logging to the SFLogging framework, which has been used with some success. Enable this on a command line by setting the property `org.apache.commons.logging.Log` to the name of the factory class, which is `org.smartfrog.services.logging.jcl.front.CommonsLogFactory`.

```
-Dorg.apache.commons.logging.Log=org.smartfrog.services.logging.jcl.front.CommonsLogFactory
```

Once this property is set, the a SmartFrog Logger for each class is chosen, usually based on the classname passed in to the Commons Logging API.

1. Do not attempt to use this logger (or this property) outside a SmartFrog daemon; it will not work, commons-logging will fail to start the application will fail with an exception.
2. Accordingly be very careful about including a `commons-logging.properties` file in any JAR on the classpath, as this will force a specific logger to be used.

6 Futures

- Possibly: a new Log4J factory that routes logs from log4j-enabled components to SFLog. This would give all Log4J-enabled components dynamic logging services
- A log that generates an Atom feed that the servlet engine can serve up (and something else to parse this and regenerate the events)

Contributions in this area are welcome.

7 Bibliography

- [ceki03] The Complete Manual for Log4J
  https://www.qos.ch/shop/products/log4j/log4j-Manual.jsp
- [log4j] Apache Log4J : http://logging.apache.org/log4j/docs/
- [jcl] Apache Jakarta Commons-Logging:
  http://commons.apache.org/logging/
- [jclts] Commons-Logging Troubleshooting:
  http://jakarta.apache.org/commons/logging/commons-logging-1.1/troubleshooting.html