Logging within SmartFrog Components

version 1.1
21th Feb, 2007

Table Of Contents

1 Introduction

SmartFrog is a tool for creating and deploying distributed systems, systems described through declarative deployment descriptors backed by a set of SmartFrog components.

A key aspect of developing and running such systems is the ability to generate, collate and inspect log files generated over the distributed environment. For this to be possible it is important to provide a simple, but powerful means for component writers to interact in a standardized way with the logging system. In this way it becomes both pervasive and consistent.

To this end, SmartFrog has a logging system as part of its fundamental core which can be extended in interesting ways, but which provides the component writer with that consistent interface common for all components and the core of SmartFrog itself.

This infrastructure is not yet fully realized, but will be over the coming period. However the component-writers view, the API, will not change over this time – merely the sophistication of the distributed collection and analysis available as part of the standard distribution.

The entire model is based on that provided by the de-facto Java standard for logging Log4J [1] and integrates with it so as to enable SmartFrog users to use the tools that come with that environment.

Note that in this documentation, there are many classes and interfaces that are described. These are, unless specified otherwise, in the package `org.smartfrog.sfcore.logging`. All of these classes are documented in their Javadoc documentation, which should be the ultimate reference material.

2 Basic Model

A SmartFrog daemon can log different applications and components through different "logs" - so that each component or set of components can deliver their logging information differently. Each such log has a unique name in that daemon, and all logging is done to a log. The programmer's API to a log is defined in the interface `LogSF`.

To create and manage these named logs, SmartFrog provides a factory to create new ones on demand, or to link to an existing one. The implementation class for this is `LogFactory`, and all methods on it are static.

Each log comes with the capability to direct log messages to a number of different destinations – *loggers*. There is a single default logging destination known as the *local logger* (not a component for technical reasons), but users can deploy logger components to direct specific log messages in various ways: to files, databases, remotely to other remote logger components, and so on.

Within a component, the writer can define the named log to use, and then all logging methods will be directed to that log, including those that come from the `PrimImpl` code that all components inherit. In most cases, the default log picked by a component is the appropriate one to use, so the component writer need do nothing but add the logging calls to that log in the component's code.

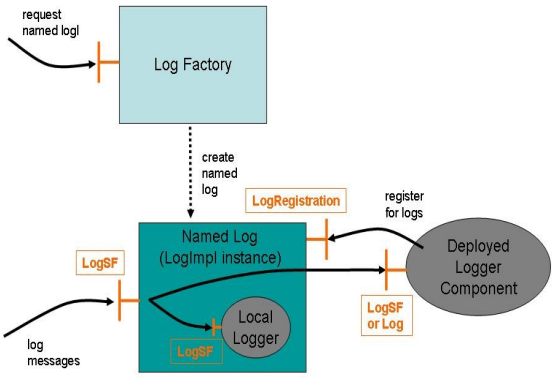

Figure 1. Daemon's Logging Infrastructure

The set of components defined in Figure 1 are now defined in more detail.

2.1 Log Factory

The log factory is a class which creates and manages logs (instances of the class `LogImpl`). A log is created with a name and two requests for the same name returns the same log. Currently, once created, a log cannot be destroyed. The methods (all static) offered by the `LogFactory` is defined in the Javadoc for the class, and are described in the section on the programmer's model.

2.2 LogSF, Log and LogMessage

The primary interface to the logging system is the `LogSF` interface. This is defined as a pure extension of the two sub-interfaces `Log` and `LogMessage`.

These two interfaces define the following:

- `Log`: the primary logging methods, the definition of logging levels, and so on
- `LogMessage`: methods to write to the standard output and standard error streams.

The full details can be obtained from the Javadoc, and they are described further in the section on the programmer's model.

2.3 LogImpl

A log is an instance of the class `LogImpl` which receives the log messages and dispatches them to the local logger and any of the logger components that have registered with it. A log implements the interface `LogSF` for use by component writers. It also implements the `LogRegistration` interface to enable logger components to register for logs.
This aspect is not yet fully implemented

2.4 Local Loggers

Logging needs to be enabled before any of the rest of SmartFrog is initialized, let alone before any components have been deployed. Consequently the core logging infrastructure cannot be built from components, but must be built from class libraries. To this end, the concept of a local logger is provided, which is an instance of a class that handles log messages by, for example, writing them to the standard output of the daemon. All logs use an instance of the same local logger class, defined at the time of SmartFrog initialization, and the log-specific instance of the local logger is created by the log at the time of its own creation.

A number of local logger classes are provided as configuration options:

- The `LogToStreams` logger, which writes the log messages to the standard out or the standard error streams of the process (and hence to the console window, for example). This is the default SmartFrog logger.
- The `LogToFile` logger which writes log messages to the specified file.
- The `LogToNothing` logger which simply ignores all the log messages.
- The `LogToLog4J` logger in the sf-loggingservices module can redirect log data to the Log4J infrastructure, using a specific Log4J configuration file or resource.

2.5 Logger Components

The long term goal is that users will be able to configure the system with a range of deployed components that register for log messages from specific logs, with specific levels of detail, and deal with the received messages as desired.

So for example, it would be possible to deploy alongside a specific application a logger that requests all the debug or trace log messages generated by that application, but none of the others. This feature could be useful during application development and testing.

However, until this is enabled in a later release, the only loggers provided are the single-class local loggers provided with every log.

3 Log Levels

Log messages generated by the core system are generated at one of a number of different levels of importance and detail. These follow the levels defined in the Log4J interface, with the addition of an “ignore” level which is treated in just about every way as trace level, and is mapped to this for propagation to loggers.

- *Ignore*: generated along with trace information, tagged with the instruction to “ignore” the log message. It is primarily used for logging the occurrence of internal *expected* exceptions that are part of normal operation. It is present so that in an emergency, even those exceptions that are normally ignored can be logged.
- *Trace*: used to provide the finest detail of system operation, it can generate a very large amount of data so use with care.
- *Debug*: intended to provide the level required to identify faults, but without the very fine detail of Trace level.
- *Info*: provides some level of useful information on normal operation, such as deployment, but without the finer levels required to follow the inner workings of the system. This is the normal level at which the system operates.
- *Warn*: more serious warnings about the state of the system, indicating some possible exceptional situation.
- *Error*: error conditions, including deployment failures, are logged at this level.
- *Fatal*: conditions that affect the stability of the application.

In addition to the system itself logging at these various levels, components may do the same, with an appropriate semantic interpretation.

Loggers are defined to work at different levels of detail, and the messages are passed to them according to the level set during their registration. In this way different levels of detail can be obtained from the different logs. If a logger requests logs of a specified level of detail, it will also receive those associated with a lower level of detail. Thus registering for info level implies that the logger will also receive warn, error and fatal log messages.

A common model of operation will be to have the local logger carry out background logging at, say, *info* level. Whilst debugging, developers could log applications at a finer level of detail such as *debug* by deploying additional logger components. As soon as the debugging process is complete, the logger components could be terminated to leave the application logging at the lower level of detail. If a problem occurs, the debug-level components can be redeployed and the system will generate the detailed log information once again.

Note that there are methods to determine the maximum log level currently required for a log to avoid generating expensive log messages which will simply be discarded. The "current" level may vary in time as logger components register and unregister from the various logs. This is different from classic Log4J operation, in which the log levels are fixed at run time and therefore can be cached in classes that log events.

4 Standard Output and Standard Error

In addition to the primary log messages, there is also a mechanism to output messages intended for the standard output or standard error of the SmartFrog daemon. These are not at a specific level, but are typically messages that one would write to the Java `System.out` and `System.err` streams. For example, the messages that appear on the SmartFrog console at startup are messages of this kind. A mechanism has been provided to handle these in a more flexible way.

The `LogSF` interface is defined as the combination of two sub-interfaces:

- `Log`, which implements the primary logging methods, levels and so on
- `LogMessage`, which implements the methods to write to the standard output and standard error streams. These methods are called `out()` and `err()` and have a number of overloaded variations. (See the Javadoc for the full description).

Each log implements the full `LogSF` interface, and this is made available to component writers. However loggers may implement either the complete interface or only the more restricted Log interface (i.e. without the out and err methods).

The log will forward calls to its out and err methods in one of two ways:

- If the logger implements the full `LogSF` interface, it forwards the calls to the logger's `out()` and `err()` methods,
- If the logger implements only the more restricted Log interface, it wraps the `out` and `err` messages as info-level `log` messages and then passes these to the logger.

All the local loggers provided as part of the distribution have the full `LogSF` interface, and they all output the message to the `System.out` and `System.err` streams. Some of them also internally wrap them as info-level messages for any other handling that is required (i.e. writing to file or passing to the Log4J infrastructure).f

5 Local Loggers

A local logger is a class that implements the `LogSF` interface. This class can be configured as a logger to be used by all logs as the way to log messages whether or not any logger components are deployed. The reason for using a local logger class in this way, rather than simply using components, is that logging is also required during the setup up the SmartFrog daemon, before components can be deployed.

Every instance of a local logger is of the same class, set at the time of SmartFrog initialization. However a new instance of the local logger class is created for each log. All configuration parameters given below are the same for each of these classes, but they are parameterized by the names of the log to which they belong. In this way, the logger instances can differentiate themselves – for example in the `LogToFile` logger by using a filename derived from the log name.

There are a number of predefined local logger classes that can be use:

- The `LogToStreams` logger, which writes the log messages to the standard out or the standard error streams of the process (and hence to the console window, for example). This is the default SmartFrog logger.
- The `LogToFile` logger which writes log messages to the specified file.
- The `LogToNothing` logger which simply ignores all the log messages.
- The `LogToLog4J` logger in the sf-loggingservices module can redirect log data to the Log4J infrastructure, using a specific Log4J configuration file or resource.

5.1 Log to streams

This logger outputs everything it gets on the output stream of the daemon (even those messages sent to err!).

The logger class for this is:

```
org.smartfrog,sfcore.logging.LogToStreamsImpl
```

The configuration attributes (see configuration section) are described in the interface `LogToErr`, and are:

- `showLogName: Boolean`
  Include the instance name in the log message?
- `showShortName: Boolean`
  Include the short name (last component) of the logger in the log message. Defaults to `true` - otherwise we'll be lost in a flood of messages without knowing who sends them.
- `showDateTime: Boolean`
  Include the current time in the log message?
- `showThreadName: Boolean`:
  Include thread name in the log message?
- `showMethodCall: Boolean`
  Include method call in the log message?
- `showStackTrace: Boolean`
  Include stack trace in log message?

- `dateFormat:String`
  Used to format times

- `errToOut: Boolean`
  Whether to redirect all err messages to the out stream to save monitoring two streams.

5.2 Log to file

The log to file local logger causes the local logger to write the log messages to a file rather than to the output stream. In addition to the attributes defined for “Log to streams”,

- `path: String`
  The directory to use for the log files

- `logFileExtension:String`
  The extension to use for the log files

- `useDatedFileName: String`
  whether to include the date in the log filename

- `useLogNameInFileName String`
  whether to name the file after the log name

- `useHostNameInFileName: Boolean`
  whether to use the hostname in the log file – especially useful if logging to an NFS mounted directory shared by many nodes

- `fileNamePrefix: String`
  a prefix to use for the filename

- `redirectSystemOutputs: Boolean`
  whether out and err messages are redirected to a file, or whether they should still be directed to the system streams.

- `append: Boolean`
  If true, then data will be written to the end of the file rather than the beginning.

5.3 Log to nothing

The log to nothing logger does exactly what would be expected – every log message is ignored, and the logger does not claim to be logging at any level. The output and error streams are still operational, however, so calls to out and err are handled appropriately. The only configuration parameter is whether the error stream should be redirected to the the output stream.

- `errToOut: Boolean`
  whether to redirect all `err()` messages to the `out()` stream to save monitoring two streams.

5.4 Log to Log4J

The LogtoLog4J feeds the output of the SmartFrog logs into Apache Log4J logging framework. This is not part of the core release of SmartFrog, but is instead provided in the sf-loggingservices module. Please consult the specific module documentation for details and examples.

6 Logger Components

Logger components are SmartFrog deployable components that register for log messages from the logging framework. These components are known as logger components, and behave in much the same way as the local loggers which are built into the system. The intention is that suers can deploy loggers for short time intervals, such as when some debugging is required, and then remove them when necessary. Although the code to support this is all present, it has not been widely tested and so the use of logger components is not supported in this version.

7 Asynchronous Logging

The semantics of the logging is that the thread that invokes the log method is the thread that delivers the log message to the loggers – within the loggers, what happens is up to the logger. For the `LogToStreams` and `LogToFile` loggers, these are also synchronous, so the thread also carries out the write statements to the streams or file respectively. This has two consequences:

1. The logs are output at the time they are generated, so there is a tight coupling between the flow of control through a component and the logs that appear, for example, on the output stream.

2. Generating logs can significantly slow up critical sections of the code.

There are times when to improve performance an asynchronous model of logging is to be preferred. With this model, log messages are queued to be handled by the loggers at some convenient time in the future without holding up the main thread of control. Of course do so may result in a failure that is notified without the logs tracing the thread of control having been output. This property can be very confusing, however there are times when it is inevitable.

To this end, the logging system provides a simple asynchronous wrapper class.

The wrapper class implements the same interfaces as a log, and is simply constructed with the log to be wrapped. The simplest way to use the wrapper is to use the following code in the component:

```
setLog(LogImplAsyncWrapper(sfLog()));
```

This will set the log returned by future calls to `sfLog()` be the asynchronous one. Note that use of both the asynchronous wrapper and direct use of the wrapped log may result in log messages that appear out of order.

Note also that the out and err methods are not made asynchronous – the wrapper simply passes through the calls to the wrapped log. The rationale for this is that these methods are normally used for debugging, or user interaction of some kind, so synchronicity is important.

8 Configuring the Logging System

The core logging infrastructure, although not built from components, is nevertheless configurable through the provision of SmartFrog descriptions. The are included with the default configuration settings in the core Jar file, `smartfrog.jar`, but these may be altered and the jar file recreated, or they may be overridden on the command line or in the properties file `default.ini`.

The primary configuration file is included with

```
#include "/org/smartfrog/sfcore/logging/logimpl.sf"
```

This file defines two attributes – the class of the local logger and the default level at which to generate logs. These, in the sf file provided as part of the system, are set to `LogToStreams` and info-level respectively.

As stated above, if there is the need to alter their values, this can be done by setting appropriate properties on the command line or in the `default.ini` file. The property names follow the normal SmartFrog convention of using the class name and attribute name combined. So these properties, if they were to be overwritten in the `default.sf` file, would be set as follows:

```
org.smartfrog.sfcore.logging.LogImpl.logLevel=3
org.smartfrog.sfcore.logging.LogImpl.loggerClass=...
```

To set them on the command line, the -D option should be used.

Each of the logger classes provided have their own configuration file setting default values, so for the `LogToStreams` logger, there is the configuration file

```
#include "/org/smartfrog/sfcore/logging/logtostreams.sf"
```

This file contains all the attributes to determine the behaviour of the logger. These attributes are defined in the description of the logger above. Again these can be overridden on the command line or in the `default.ini` file using a similar attribute naming scheme:

```
org.smartfrog.sfcore.logging.LogToStreamsImpl....
```

Examples of how to set all these attributes are provided in the `default.sf` provided as part of the release of SmartFrog.

The Log4J logger is configured slightly differently, as amongst other things a Log4J configuration file or resource must also be provided. This is covered in the sf-loggingservices module documentation

8.1 Component-specific Configuration

To configure a single component's logging, then an `SFLog` instance needs to be deployed to set up the logging:

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

sfConfig extends Compound {
  log extends SFLog {
    logFrom LAZY PARENT:myapp;
    logTo extends LogToLog4JImpl {
      resource "log4j.properties";
    }
  }
  myapp extends MyCustomApplication {
   ...
  }
}
```

This deploys an SFLog instance that binds the `myapp` component to log via Log4J. In other deployments, the `myapp` application will not log this way, so making logging setup an option which can be tuned on a deployment-by-deployment basis.

9 Logging Futures

The logging services of SmartFrog will continue to evolve, to meet the needs of users of the framework. Without committing to any specific roadmap or schedule, here are some possible improvements we are (as of Feburary 2007) considering:

- A relay from SFLogs to logging components.
- A back end for commons-logging. This would let libraries written against the Apache Commons-Logging API to log to the SmartFrog logging framework.
- A back end for the Java1.4 Logging framework. Very low priority.
- A serializable log format for remote logging. Development on this is being driven by the test framework.
- An Atom log source, possibly as a Log4J appender. This would save log information to XML files that could be served up to RSS viewer applications, or processed through products such as Yahoo! Pipes.
- An XMPP log source. This would relay events of a specific level, such as *error* or *fatal*, to a recipient listed over XMPP. This could be used for push-model notifications.

As SmartFrog is developed as an open source project, all such improvements are dependent upon the contributions of willing developers. Anyone who has need for these features or other logging-related services, is strongly encouraged to participate in the project's development.

10 Logging within a Component

Logging is a very effective way of providing debugging services to component users, as it can be turned on production systems, and dynamically manipulated. With the Log4J logger set up to output HTML files, and the Jetty component set to serve up the directory where these are created, the log files can be remotely accessed.

To use a log, the component developers must:

1. Acquire the log
2. (possibly) set the log level
3. submit messages to the log at the appropriate levels
4. (possibly) send messages to the out and err streams

These are considered in turn:

10.1 Acquiring the log

A method is provided in `PrimImpl` which returns a log for components to use.

```
LogSF sfLog();
```

This method does the following.

1. If a log has been set by the method `setLog(...)`, this is returned.
2. If this has not been called explicitly, then the `sfLog` method generates a name on first use by doing one of the following:

1. It examines the context of the component to see the attribute `sfLog` has been set, and if so returns the log whose name is given as the value of that attribute.

2. If this does not exist, it tries to identify the log of the parent component.

3. If there is no parent component (that is, it is the root component) the name of that component is used as returned by `sfCompleteName()`.

4. If there is any difficulty in obtaining this, the daemon's core log (the log of the process compound) is used, guaranteeing that logs can always be generated.

In this way, unless specifically overridden by case 1 or 2.1, and assuming that there are no difficulties, every component in a component hierarchy will use the log with the same name as the root component. Note that this may not actually be the same instance of a log as the components may be in different daemons.

However if some other mechanism is to be used to generate a log, the log factory must be used directly, and then set as the log by using the `setLog(...)` method.

The interface to the log factory includes the following method to access a named log:

```
/**
 * get a named log.
 * @param name
 * @return a log from cache or new.
 *
 */
public static synchronized LogSF getLog(String name);
```

This method can be called from inside the body of a component, then registered as the component log, as in:

```
LogSF myLog = LogFactory.getLog("foo");
setLog(myLog);
```

It is normal to use the `sfLog()` method to obtain the desired component log rather than store it oneself. This gives a uniform way to redirecting the entire component's logging by using a single `setLog()` call. However if the situation arises that a component needs to send messages to many different logs, these will have to be created and managed explicitly by that component.

10.1.1 When to call sfLog()

You cannot call `sfLog()` in the constructor of an object. Only use it (and hence, start logging) in `sfDeploy()` or later lifecycle events.

10.2 Setting the log level

Under normal circumstances, the logging is set through configuration parameters in the system setup as described in the section on the configuration of the logging. The default level set up in this way is the info level. This level is in fact the level of the local logger, the logger components may each have their own level and the current logging level for the user of the log is the most prolific (detailed) of the levels.

It is possible, though not normally recommended, for a component to set the logging level of the local logger through the log. This affects the logging level of all components that use the same log; this is normal behaviour. However it can occasionally useful to include a specialized component whilst debugging that alters the log level of the applications log to a more detailed level. A better approach, when supported, will be to add a new component logger which registers for a more detailed level.

The methods for reading and altering the log's level are:

```
public void setLevel(int level);
public int getlevel();
```

The levels are 0 for ignore, through to 6 for fatal. Constants for these are provided in the `LogLevel` interface.

10.3 Submitting log messages

A component can log whenever it feels it must, once the component is deployed. The exact style should be familiar to anyone who has used the commons-logging or Log4J APIs:

```
sfLog().info("starting");
try {
   execute();
} catch(Exception exception) {
  sfLog.error("Failed to execute "+commandLine,exception);
}
```

Here we have logged a startup message at the info level, then, on an error, logged the exception along with more detailed text. It is always a good practise to provide more verbose output during failures, as that is the time that log matters the most.

If the cost of constructing a message is high, component execution times can be improved by skipping the logging process when a log level is not required:The standard pattern for this is:

```
if (sfLog().isDebugEnabled()) {
        StringBuffer message = new StringBuffer()
        // construct the message
        sfLog().debug(message.toString());
}
```

This pattern works for each of the logging levels mentioned above. As the default level of logging is `info`, it is reasonable to only check before logging at a more detailed level. Furthermore, the cost saving is not the method call, here `sfLog().debug()`., as the execution time of `sfLog().isDebugEnabled()` will be equivalent. What is being saved is the string creation/concatenation overhead. If only a static string is being logged, there is no need to check the log level.

10.4 Outputting to standard out and standard error

As discussed earlier, the logging has been enhanced with the ability to send messages directly to the output or error streams of the daemon.

This is possible using the `System.out` and `System.err` streams in Java, however so as to allow these to be logged or specially handled in some way, a mechanism has been added to each log.

The interface, `LogMessage` and its extension `LogSF`, provide a number of methods which are described in more detail in the Javadoc for the interfaces. However for illustration, the following code snippets may be useful.

```
String message = ...;
sfLog().out(message);

try {
    ....;  // some code
} catch (Exception e) {
    sfLog().err("trouble", e);
}
```

The err method has a number of overloaded variants, including with and without a `Throwable` parameter, and with and without a termination record parameter. The full details are in the Javadoc for the interface `LogMessage`.

10.5 Component Developer best practices

Here are some recommendations to component developers to produce components with good logging.

1. Do not print messages to `System.out` or `System.err`; use the logging API to log events.
2. Add copious logging, especially at the debug and trace level.
3. Use `sfLog.isDebugEnabled()` to skip time-consuming string construction operations.
4. Do not go overboard in logging at the info level, as it generates a noisy log.
5. Use the `error` level for logging errors.
6. Leave the `fatal` level for situations in which SmartFrog itself is likely to fail. Components are not expected to create this.
7. If you swallow an exception, even if you think it is utterly impossible it will ever be raised, log it at the `ignored` level. That way, when it does surface, people can find out what is going wrong without editing your code.
8. Include the exception itself when logging exceptions, as it may contain detailed diagnostics.
9. Don't cache the log level, as it may change dynamically.
10. Try not to do things that throw exceptions in the logging messages themselves. Calling methods on null pointers is a classic example. The following is wrong
    `sfLog.error("project="+project.toString());`
    It is safer to rely on the string concatenation tool to detect the null pointer and print (null):
    `sfLog.error("project="+project);`
    or, especially for other methods, to check for null parameters first:
    `sfLog.error("project="+project==null?"Null project":project.status());`
11. Remember that most SmartFrog components are intended for automated deployment -do not assume that there is anyone present to read the log. Logging messages should be a secondary activity. In particular, if something goes wrong for which action is required, throw an exception instead of just logging it

12. If you intend to use the test tools to capture the log -and make assertions about it- stick the messages into constant string fields in the class, strings that Java-based test cases can link to. This stops tests breaking when a message is changed.

13. Do not under any circumstances include a /log4.properties resource in your module's JAR file. This may interfere with custom versions of the file created for specific installations. If you use Log4J to log in your application, place example loggers into your specific Java packages. Adding a version of /log4.properties is acceptable in the test source tree, because that is not normally redistributed.

11 Summary

The SmartFrog logging services enable components to be instrumented with extra logging.