Inq and Logging

Mini Guide

Table of Contents

Document History

Version	Date	Notes
0.1	04/04/10	Initial Draft
0.2	31/05/10	Added property and log formatter extensions

1 Introduction

This document discusses the logging functionality available in Inq. There are two popular logging systems in general use by the Java community. These are Apache log4j and `java.util.logging`, bundled with the JRE. The consensus seems to be that log4j (sometimes accessed through the harmonising layer of Apache commons logging) is overall the better alternative, based on ease of configuration and flexibility of log destination and format. However within the Inq environment, in this area like others, such flexibility is delivered by calling into an Inq function where arbitrary script can be executed. For example, that log4j includes `SMTPAppender` as an email interface can be supported in Inq by attaching a call to a logging function that invokes the Inq `sendmail` primitive. As Inq is extended for example with JMS access, so other log sinks implicitly become available. Furthermore, at present Inq depends only on JRE 1.6 or better and for these reasons the underlying log functionality is provided by `java.util.logging`.

Certain extensions have been added to support logger-based configuration of handler properties, a substitution syntax within the logging configuration file and a log formatter that produces messages suitable for a multi-threaded environment. These further mitigate the limitations of Java logging over log4j.

This document assumes the reader is familiar with Inq in general. Further information on the Inq language is available at the Inqwell website. The Java logging package is described at JavaSoft

2 Configuring Logging

The documentation for `java.util.logging` details logging configuration. Inq provides no means of initialising the logging environment although script can modify certain aspects, such as levels and the use of Inq logging functions, via property access to the prevailing logging entities. In the remainder of this document the implicit configuration will be that defined by `$INQHOME/etc/log.properties`.

2.1 Setting The System Properties

The `java.util.logging` package reads its configuration by defining the `java.util.logging.config.file` system property, or by using any of the other available methods supported by Java. Without further effort the named loggers are available for use from Inq script:

```
logmessage("foo.test", LOG_INFO, "An information message");
```

Inq logging is not fully enabled unless the `java.util.logging.manager` system property is set to the value `com.inqwell.any.AnyLogManager`. With this property set it is possible to place a named logger into the node space and thus refer to it indirectly in `logmessage` statements:

```
any $this.vars.logger = getlogger("foo.test");
       .
       .
logconfig($this.vars.logger, "A config message");
```

The launch scripts in the Inq distribution establish the log manager by defining the system property on the command line: `-Djava.util.logging.manager=com.inqwell.any.AnyLogManager`.

2.2 Inq Logger Property Access

With `AnyLogManager` in effect, putting a logger into the node space allows its properties to be accessed. A

logger supports the following properties:

- `logLevel` the level at which log messages must be at or above in order to actually be logged.
- `logHandlers` the handlers placed on this logger. Inq does not define any handlers of its own but allows access to any that have been configured. A logger can have more than one handler so this property is an array of `Handlers`.
- `useParentHandlers` whether this logger logs messages to its parent.
- `logFunc` a `call` statement which, if established acts as a Filter for the logger. The target function is passed the logging record as individual parameters and can perform additional logging functionality, such as sending email. The `logFunc` Filter is discussed further below.
- `logFilter` the Filter placed on this logger. If a `logFunc` has been established then this will be the Inq Filter implementation that calls the function.

2.3 Inq Configuration Enhancements

A major limitation of the Java logging system is its inability to configure handlers on a per-logger basis. Furthermore, apart from limited capabilities of specific handlers, such as `%t` specifying the platform temporary directory in the `java.util.logging.FileHandler.pattern` property, there is no support for symbolic expansion of property values.

2.3.1 Per-Logger Properties

With the Inq `AnyLogManager` in effect, the following properties relevant to various handlers are supported as suffixes to their logger name: `.pattern`, `.limit`, `.count`, `.append`, `.formatter`, `.encoding`, `.filter`, `.host` and `.port`. When used as a logger property and relevant to the Handler being applied, properties like `<logger>.pattern` can be specified to configure only that logger's use of the `Handler` class (the `FileHandler` class in this case).

2.3.2 Symbolic Property Values

Definitions in the logger properties file are simple values with no general method of substitution. For example, the `pattern` property of the `FileHandler` class offers only limited scope to configure the directory where the log files will be placed.

When the Inq `AnyLogManager` is in effect, all property values when queried by the Java logging system support a `${foo}` substitution syntax. The token `foo` is taken as a property key and queried first in the logging properties and then, if not resolved, in the system properties. Consider the following logging properties fragment:

```
eek=xyz
 .
 .
inq.filexfer.pattern=${inq.home}/log/${eek}.log
```

When the property `inq.filexfer.pattern` is requested the sequence `${inq.home}` is substituted from the System property `inq.home` and `${eek}` from the logging property `eek`. The resulting value might then be `/home/tom/inqwell/dev/log/xyz.log`.

3 Logging Usage

3.1 Log A Message

The syntax of `logmessage` is as follows:

```
“logmessage” “(“ <logger-reference> “,”
                <logging-level> “,”
                <message>
                [ ( “,” <parameter>  )… ]
            “)”
```

where all arguments are expressions.

The level constants in ascending order of severity are `LOG_FINEST`, `LOG_FINER`, `LOG_FINE`, `LOG_CONFIG`, `LOG_INFO`, `LOG_WARNING` and `LOG_SEVERE`. There are corresponding short-hand functions that imply these levels, for example

```
logconfig(“foo.test”, “A config message”);
```

The following functions have the same arguments excepting `<logging-level>` which is implicit: `logfinest()`, `logfiner()`, `logfine()`, `logconfig()`, `loginfo()`, `logwarning()` and `logsevere()`. Any number of optional `<parameter>` arguments may be specified.

When referring to a logger, the `<logger-reference>` argument can be a string representing a logger name or a reference to a logger held in the node space. Note that holding a reference to a previously retrieved logger is preferable – loggers are eligible for garbage collection if not held in this way. Configured loggers are always available but reinitialising a logger will reinitialise its handlers, possibly rolling log files over earlier than expected.

Log messages are formatted by Formatter implementations. The standard `SimpleFormatter` and `XMLFormatter` classes process the parameters into the log output according to their Javadoc. Inq provides the `com.inqwell.any.logging.SimpleFormatter` class that formats log messages as follows:

- A log entry occupies a single line (unless the message contains a line separator)
- The log time format includes milliseconds
- The log entry includes the name of the thread

An example message looks like this:

```
[31-May-2010 21:11:04.455 CONFIG] This is a log message
(file:///home/tom/inqwell/dev/app/filexfer/FileXfer.inq:43 inq.filexfer:initialise in thread
"main")
```

Note that the URL of the script where the log message originates, the line number and the fully-qualified name of the executing function are included.

The `com.inqwell.any.logging.SimpleFormatter` class is not currently configurable in any way.

3.2 Using Logging Levels

Loggers and handlers support a logging level. A logger can have multiple handlers – if a log message is at

a level that defines it as loggable, it may still not be logged by one or more of the handlers, depending on their specified level.

Default levels for normal system operation will be established in logging configuration. Generally, `LOG_INFO` is chosen and `loginfo()` statements represent normal logging output. Any log messages at a higher level will also be logged. Information considered less important can be logged at a lower levels and then enabled by altering the level of the appropriate logger or handler. To alter the level of a logger, first fetch it by name and then access its `logLevel` property:

```
any logger = getlogger("foo.test");
logger.properties.logLevel = LOG_FINE;
```

If a logger has more than one handler then log messages are dispatched to each in turn. Since each handler has an associated Formatter, handlers can be used to produce log output in a variety of formats. Alternatively, separate handlers can be used to keep the principle log sink clear of messages of lower importance while allowing another to receive them. The handlers placed on a logger are not named, so are returned as an array. They can then be accessed in the order they are defined in the configuration:

```
any logger = getlogger("foo.test");
any handlers = logger.properties.logHandlers;
handler[0].properties.logLevel = LOG_FINE;
```

The level `LOG_ALL` means that all messages will be logged by the logger or handler while `LOG_OFF` disables logging altogether.

3.3 Defining A Filter Function

A logger and handler both support a single filter. In the Inq environment a filter is placed on a logger or handler by setting the `logFunc` property. If a message is eligible for logging according to its level the filter function will be called. This function can perform any scripted actions and returns `true` to log the message and `false` to veto it.

Here is an example of a log function:

```
// A function for the logger to call
local function exampleLogFunc(any level,
                              any loggerName,
                              any message,
                              any atTime,
                              any seq,
                              any sourceUrl,
                              any sourceFunc,
                              any procName,
                              any params)
{
  // Just show the stack so we see all the arguments.
  writeln($catalog.system.out, .);

  // Test the level for something horrendous
  if (level >= LOG_SEVERE)
    writeln($catalog.system.out, "HELP send an email!");

  // Always log the message
  true;
}
```

```
any logger = getlogger(“foo.test”);

// Apply the log function
logger.properties.logFunc = cfunc f = call exampleLogFunc();

// log a message
logmessage(logger,
           LOG_SEVERE,
           "Log message 4 1st param is {0} second param is {1}", "p1", "p2");
```

The output will be something like:

```
{message=Log message 4 1st param is {0} second param is {1}, sourceFunc=<parser>, level=SEVERE,
seq=2, params=[p1, p2], atTime=Mon May 31 21:09:39 BST 2010,
sourceUrl=file:////home/tom/inqwell/dev/examples/logging.inq:51, loggerName=foo.test}
HELP send an email!
[31-May-2010 21:09:39.129 SEVERE] Log message 4 1st param is p1 second param is p2
(file:////home/tom/inqwell/dev/examples/logging.inq:51 <parser> in thread "main")
```

3.4 Log Message Origin

Notice in the above example that the log message includes the source URL and Inq function (the parser in this case) in which the `logmessage` statement appears. This is analogous to the class and method names in Java logging. These, as well as all other log record aspects are passed to any logging function. Note that `AnyLogManager` must be in effect for the log message origin to be correctly reported.