DevNote #103 – How to Customize Logging in the ADK

The SIFWorks ADK for Java includes a logging facility that is based on the open-source Log4j project from the Apache Software Foundation. Log4j employs a lightweight, flexible architecture in which log entries are organized into Categories and output to one or more Appenders. When the ADK is initialized, it establishes three logging Categories to which all log messages are written: one for the ADK itself, one for your Agent class, and one for each Zone instance that's created by your agent. The default Log4j ConsoleAppender is used to direct log output from these three Categories to System.out.

This Tech Note shows how to customize logging in the ADK to use your own Log4j Appender implementation. You can also download and use a variety of Appenders that have been written by other developers for logging to the file system, to the Windows NT Event Log, to SQL databases, and to SNMP.

Download third-party Log4j Appender implementations from:

http://jakarta.apache.org/log4j/docs/download.html

For more information on developing your own Appender implementation:

http://jakarta.apache.org/log4j/docs/documentation.html

NOTE: A sample agent illustrating the concepts in this DevNote is available for download from the Edustructures Customer Support website. Once signed in to the site, click on the Support Forums link. Or, browse the Documentation & Tech Notes section for the ADK.

1. Accessing the ADK's Log Categories

The ADK establishes three Log4j Categories to which all log events are written:

- Global ADK Category
- Agent Category
- Zone Category (one per Zone instance)

If you want to contribute to the ADK's logging you can obtain a reference to any of these Category objects and then call its public methods to output a log message.

To access the global ADK Category:

Call the `ADK.getLog()` static method:

```
public static Category getLog();
```

The global ADK Category is used by the class framework to log messages that occur before your Agent class is constructed.

To access the Agent Category:

Call the static `Agent.getLog()` method once your Agent instance is constructed:

```
public static Category getLog();
```

The Agent Category is typically used by the class framework to log messages that are not associated with any one zone.

To access a Zone Category:

Call the `Agent.getLog( Zone )` method once a Zone instance is constructed:

```
public Category getLog( Zone zone );
```

Zone Categories are used by the class framework to log messages that are associated with a specific zone. Since the majority of work that takes place in the ADK is the result of SIF messaging, most log entries are written to zone Categories.

2. Writing Log Entries to a Category

Contributing to the ADK's log Categories is easy:

```
Agent.getLog().error( "The agent cannot start up because of an error: " + message );
```

NOTE: The ADK uses the DEBUG log level for all of its log messages.

3. Changing the Default Log4j Appender

When the ADK is initialized, it configures a default ConsoleAppender for each of the above Categories so that all log output will be written to System.out. While this is an appropriate default behavior for a library, most agent developers will want to redirect log output to a file, to a database, or to a window in the agent's administration console. This is easily done by changing the Appender configuration of each Category.

This example shows how to customize the Agent Category to use the RollingFileAppender that's included with Log4j:

```
public class MyAgent extends Agent
{
   public void initialize()
   {
      super.initialize();

      // Get the Agent's log Category
      Category logger = getLog();

      // Create a DailyRollingFileAppender to write to agent.log and roll it daily
      DailyRollingFileAppender fileLog = new DailyRollingFileAppender(
         new PatternLayout( ADK.DEFAULT_LOG4J_PATTERN ),
         "agent.log",
         "'.'yyyy-MM-dd" );

      // Add this Appender to the log Category
      logger.addAppender( fileLog );

      ...
   }
```

With Log4j, you can add as many Appenders to a Category as needed and log output will be written to each. If you only want one Appender to be active, be sure to clear out the ADK's default before calling the Category.addAppender method:

```
      // Remove the ADK's default System.out Appender
      logger.removeAllAppenders();
```

```
        // Add the DailyRollingFileAppender
        logger.addAppender( fileLog );
```

You can apply the same code illustrated above to each zone your agent is connected to. Because each Zone instance is assigned its own Category by the ADK, it is easy to redirect zone-specific output to individual files rather than grouping all zones in a single log file. In the example below, each zone's log output will be written to a separate directory having the name of the zone (e.g. "c:\myagent\logs\ZONE_1\zone.log")

```
public class MyZoneClass
{
   public MyZoneClass( Agent agent, Zone zone )
   {
      // Get the log Category for this Zone
      Category logger = Agent.getLog( zone );

      // The zone log file is: agent-directory/logs/zoneId/zone.log
      String filename = agent.getHomeDir() + File.separator + "logs" + File.separator +
         zone.getZoneId() + File.separator + "zone.log";

      // Create a DailyRollingFileAppender to write to zoneId.log file and roll it daily
      DailyRollingFileAppender fileLog = new DailyRollingFileAppender(
         new PatternLayout( ADK.DEFAULT_LOG4J_PATTERN ),
         filename,
         "'.'yyyy-MM-dd" );

      // Add this Appender to the log Category
      logger.addAppender( fileLog );

      ...
   }
}
```

4. Adjusting the Log Level

To adjust the amount of logging produced by the ADK, you can modify the value of the static `ADK.debug` variable at any time. For example,

```
// Turn off all logging except critical exceptions and agent startup/shutdown
ADK.debug = ADK.DBG_EXCEPTIONS | DBG_LIFECYCLE;

// Turn on moderate logging (see table below)
ADK.debug = ADK.DBG_MODERATE;
```

The ADK class defines several `DBG_` flags that specify which kinds of activity should be written to the log. By default all are enabled except `DBG_MESSAGE_CONTENT`. You can combine these flags using the logical OR operator (e.g. "`( DBG_TRANSPORT | DBG_MESSAGING | DBG_PROVISIONING )`":

Flag	Value	Meaning
DBG_TRANSPORT	0x00000004	Log HTTP and HTTPS transport-related activity
DBG_MESSAGING	0x00000008	Log incoming and outgoing message information, but not message content or Pull requests
DBG_MESSAGING_PULL	0x00000040	Log Pull requests
DBG_MESSAGING_DETAILED	0x00000080	Log detailed SIF Header message information
DBG_MESSAGING_CONTENT	0x00000100	Log the XML content of SIF messages
DBG_PROVISIONING	0x00000200	Log the details of provisioning messages (e.g. SIF_Register, SIF_Provide, SIF_Subscribe, etc.)
DBG_LIFECYCLE	0x10000000	Log agent startup and shutdown activity
DBG_EXCEPTIONS	0x20000000	Log exceptions thrown by the ADK
DBG_PROPERTIES	0x40000000	Log the value of AgentProperties specified from the –D Java command-line option (useful for verifying that the options you specify with –D are set at runtime.)

These preset combinations are also defined:

Flag	Value
DBG_ALL	0xFFFFFFFF
DBG_NONE	0
DBG_MINIMAL	DBG_EXCEPTIONS | DBG_PROVISIONING
DBG_MODERATE	DBG_MINIMAL | DBG_MESSAGING | DBG_LIFECYCLE
DBG_MODERATE_WITH_PULL	DBG_MODERATE | DBG_MESSAGING_PULL
DBG_DETAILED	DBG_MODERATE_WITH_PULL | DBG_TRANSPORT | DBG_MESSAGING_DETAILED
DBG_VERY_DETAILED	DBG_DETAILED | DBG_MESSAGING_EVENT_DISPATCHING | DBG_MESSAGING_RESPONSE_PROCESSING | DBG_PROPERTIES