Chapter 10. Logging Conventions

Persisted diagnostic logs are often very useful in debugging software issues. This section lists some general guidelines followed in JBoss code for diagnostic logging.

10.1. Obtaining a Logger

The following code snippet illustrates how you can obtain a logger.

```
package org.jboss.X.Y;
import org.jboss.logging.Logger;

public class TestABCWrapper
{
   private static final Logger log = Logger.getLogger(TestABCWrapper.class.getName());

   // Hereafter, the logger may be used with whatever priority level as appropriate.
}
```

After a logger is obtained, it can be used to log messages by specifying appropriate priority levels.

10.2. Logging Levels

1. FATAL - Use the FATAL level priority for events that indicate a critical service failure. If a service issues a FATAL error it is completely unable to service requests of any kind.
2. ERROR - Use the ERROR level priority for events that indicate a disruption in a request or the ability to service a request. A service should have some capacity to continue to service requests in the presence of ERRORs.
3. WARN - Use the WARN level priority for events that may indicate a non-critical service error. Resumable errors, or minor breaches in request expectations fall into this category. The distinction between WARN and ERROR may be hard to discern and so its up to the developer to judge. The simplest criterion is would this failure result in a user support call. If it would use ERROR. If it would not use WARN.
4. INFO - Use the INFO level priority for service life-cycle events and other crucial related information. Looking at the INFO messages for a given service category should tell you exactly what state the service is in.
5. DEBUG - Use the DEBUG level priority for log messages that convey extra information regarding life-cycle events. Developer or in depth information required for support is the basis for this priority. The important point is that when the DEBUG level priority is enabled, the JBoss server log should not grow proportionally with the number of server requests. Looking at the DEBUG and INFO messages for a given service category should tell you exactly what state the service is in, as well as what server resources it is using: ports, interfaces, log files, etc.
6. TRACE - Use TRACE the level priority for log messages that are directly associated with activity that corresponds requests. Further, such messages should not be submitted to a Logger unless the Logger category priority threshold indicates that the message will be rendered. Use the `Logger.isTraceEnabled()` method to determine if the category priority threshold is enabled. The point of the `TRACE` priority is to allow for deep probing of the JBoss server behavior when necessary. When the `TRACE` level priority is enabled, you can expect the number of messages in the JBoss server log to grow at least a x N, where N is the number of requests received by the server, a some constant. The server log may well grow as power of N depending on the request-handling layer being traced.

10.3. Log4j Configuration

The log4j configuration is loaded from the jboss server conf/log4j.xml file. You can edit this to add/change the default appenders and logging thresholds.

10.3.1. Separating Application Logs

You can segment logging output by assigning log4j categories to specific appenders in the conf/log4j.xml configuration.

Example 10.1. Assigning categories to specific appenders

```
 <appender name="App1Log" class="org.apache.log4j.FileAppender">
    <errorHandler
         class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
    <param name="Append" value="false"/>
    <param name="File"
         value="${jboss.server.home.dir}/log/app1.log"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
           value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
    </layout>
  </appender>

...

  <category name="com.app1">
    <appender-ref ref="App1Log"/>
  </category>
  <category name="com.util">
    <appender-ref ref="App1Log"/>
  </category>

...

  <root>
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
    <appender-ref ref="App1Log"/>
  </root>
```

10.3.2. Specifying appenders and filters

If you have multiple apps with shared classes/categories, and/or want the jboss categories to show up in your app log then this approach will not work. There is a new appender filter called TCLFilter that can help with this. The filter should be added to the appender and it needs to be specifed what deployment url should logging be restricted to. For example, if your app1 deployment was app1.ear, you would use the following additions to the conf/log4j.xml:

Example 10.2. Filtering log messages

```
  <appender name="App1Log" class="org.apache.log4j.FileAppender">
    <errorHandler
        class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
    <param name="Append" value="false"/>
    <param name="File"
         value="${jboss.server.home.dir}/log/app1.log"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
           value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
```

```
    </layout>
    <filter class="org.jboss.logging.filter.TCLFilter">
      <param name="AcceptOnMatch" value="true"/>
      <param name="DeployURL" value="app1.ear"/>
    </filter>
  </appender>

...

  <root>
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
    <appender-ref ref="App1Log"/>
  </root>
```

10.3.3. Logging to a Seperate Server

The log4j framework has a number of appenders that allow you to send log message to an external server. Common appenders include:

1. org.apache.log4j.net.JMSAppender
2. org.apache.log4j.net.SMTPAppender
3. org.apache.log4j.net.SocketAppender
4. org.apache.log4j.net.SyslogAppender
5. org.apache.log4j.net.TelnetAppender

Documentation on configuration of these appenders can be found at Apache Logging Services.

JBoss has a Log4jSocketServer service that allows for easy use of the SocketAppender.

Example 10.3. Setting up and using the Log4jSocketServer service.

The org.jboss.logging.Log4jSocketServer is an mbean service that allows one to collect output from multiple log4j clients (including jboss servers) that are using the org.apache.log4j.net.SocketAppender.

The Log4jSocketServer creates a server socket to accept SocketAppender connections, and logs incoming messages based on the local log4j.xml configuration.

You can create a minimal jboss configuration that includes a Log4jSocketServer to act as your log server.

Example 10.4. An Log4jSocketServer mbean configuration

The following MBean Configuration can be added to the conf/jboss-service.xml

```
  <mbean code="org.jboss.logging.Log4jSocketServer"
    name="jboss.system:type=Log4jService,service=SocketServer">
    <attribute name="Port">12345</attribute>
    <attribute name="BindAddress">${jboss.bind.address}</attribute>
  </mbean>
```

The Log4jSocketServer adds an MDC entry under the key 'host' which includes the client socket InetAddress.getHostName value on every client connection. This allows you to differentiate logging output based on the client hostname using the MDC pattern.

Example 10.5. Augmenting the log server console output with the logging client socket hostname

```
    <appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
```

```
    <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
    <param name="Target" value="System.out"/>
    <param name="Threshold" value="INFO"/>

    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1},%X{host}] %m%n"/>
    </layout>
  </appender>
```

All other jboss servers that should send log messages to the log server would add an appender configuration that uses the SocketAppender.

Example 10.6. log4j.xml appender for the Log4jSocketServer

```
<appender name="SOCKET" class="org.apache.log4j.net.SocketAppender">
  <param name="Port" value="12345"/>
  <param name="RemoteHost" value="loghost"/>
  <param name="ReconnectionDelay" value="60000"/>
  <param name="Threshold" value="INFO"/>
</appender>
```

10.3.4. Key JBoss Subsystem Categories

Some of the key subsystem category names are given in the following table. These are just the top level category names. Generally you can specify much more specific category names to enable very targeted logging.

Table 10.1. JBoss SubSystem Categories

SubSystem	Category
Cache	org.jboss.cache
CMP	org.jboss.ejb.plugins.cmp
Core Service	org.jboss.system
Cluster	org.jboss.ha
EJB	org.jboss.ejb
JCA	org.jboss.resource
JMX	org.jboss.mx
JMS	org.jboss.mq
JTA	org.jboss.tm
MDB	org.jboss.ejb.plugins.jms, org.jboss.jms
Security	org.jboss.security
Tomcat	org.jboss.web, org.apache.catalina
Apache Stuff	org.apache
JGroups	org.jgroups

10.3.5. Redirecting Category Output

When you increase the level of logging for one or more categories, it is often useful to redirect the output to a seperate file for easier investigation. To do this you add an appender-ref to the category as shown here:

Example 10.7. Adding an appender-ref to a category

```
  <appender name="JSR77" class="org.apache.log4j.FileAppender">
    <param name="File"
          value="${jboss.server.home.dir}/log/jsr77.log"/>
...
  </appender>

  <!-- Limit the JSR77 categories -->
  <category name="org.jboss.management" additivity="false">
    <priority value="DEBUG"/>
    <appender-ref ref="JSR77"/>
  </category>
```

This sends allorg.jboss.management output to the jsr77.log file. The additivity attribute controls whether output continues to go to the root category appender. If false, output only goes to the appenders referred to by the category.

10.3.6. Using your own log4j.xml file - class loader scoping

In order to use your own log4j.xml file you need to do something to initialize log4j in your application. If you use the default singleton initialization method where the first use of log4j triggers a search for the log4j initialization files, you need to configure a ClassLoader to use scoped class loading, with overrides of the jBoss classes. You also have to include the log4j.jar in your application so that new log4j singletons are created in your applications scope.

> **Note**
>
> You cannot use a log4j.properties file using this approach, at least using log4j-1.2.8 because it preferentially searches for a log4j.xml resource and will find the conf/log4j.xml ahead of the application log4j.properties file. You could rename the conf/log4j.xml to something like conf/jboss-log4j.xml and then change the ConfigurationURL attribute of the Log4jService in the conf/jboss-service.xml to get around this.

10.3.7. Using your own log4j.properties file - class loader scoping

To use a log4j.properties file, you have to make the change in conf/jboss-service.xml as shown below. This is necessary for the reasons mentioned above. Essentially you are changing the log4j resource file that jBossAS will look for. After making the change in jboss-service.xml make sure you rename the conf/log4j.xml to the name that you have give in jboss-service.xml (in this case jboss-log4j.xml).

```
<!--=====================================================================-->
<!-- Log4j Initialization                                -->
<!-=====================================================================-->

<mbean code="org.jboss.logging.Log4jService"
  name="jboss.system:type=Log4jService,service=Logging">
  <attribute name="ConfigurationURL">
          resource:jboss-log4j.xml</attribute>
 <!-- Set the org.apache.log4j.helpers.LogLog.setQuiteMode.
  As of log4j1.2.8 this needs to be set to avoid a possible deadlock
  on exception at the appender level. See bug#696819.
 -->
 <attribute name="Log4jQuietMode">true</attribute>
```

```
 <!-- How frequently in seconds the ConfigurationURL is checked for changes -->
 <attribute name="RefreshPeriod">60</attribute>
</mbean>
```

Drop log4j.jar in your myapp.war/WEB-INF. Make the change in jboss-web.xml for class-loading, as shown in the section above. In this case, myapp.war/WEB-INF/jboss-web.xml looks like this:

```
<jboss-web>
  <class-loading java2ClassLoadingCompliance="false">
  <loader-repository>
     myapp:loader=myapp.war
     <loader-repository-config>java2ParentDelegation=false
     </loader-repository-config>
   </loader-repository>
 </class-loading>
</jboss-web>
```

Now, in your deploy/myapp.war/WEB-INF/classes create a log4j.properties.

Example 10.8. Sample log4j.properties

```
# Debug log4j
log4j.debug=true
log4j.rootLogger=debug, myapp

log4j.appender.myapp=org.apache.log4j.FileAppender
log4j.appender.myapp.layout=org.apache.log4j.HTMLLayout
log4j.appender.myapp.layout.LocationInfo=true
log4j.appender.myapp.layout.Title='All' Log
log4j.appender.myapp.File=${jboss.server.home.dir}/deploy/myapp.war/WEB-INF/logs/myapp.html
log4j.appender.myapp.ImmediateFlush=true
log4j.appender.myapp.Append=false
```

The above property file sets the log4j debug system to true, which displays log4j messages in your jBoss log. You can use this to discover errors, if any in your properties file. It then produces a nice HTML log file and places it in your application's WEB-INF/logs directory. In your application, you can call this logger with the syntax:

```
...
private static Logger log = Logger.getLogger("myapp");
...
log.debug("############## A debug message from myapp logger #########");
...
```

If all goes well, you should see this message in myapp.html.

After jBossAS has reloaded conf/jboss-service.xml (you may have to restart jBossAS), touch myapp.war/WEB-INF/web.xml so that JBoss reloads the configuration for your application. As the application loads you should see log4j debug messages showing that its reading your log4j.properties. This should enable you to have your own logging system independent of the JBoss logging system.

10.3.8. Using your own log4j.xml file - Log4j RepositorySelector

Another way to achieve this is to write a custom RepositorySelector that changes how the LogManager gets a logger. Using this technique, Logger.getLogger() will return a different logger based on the

context class loader. Each context class loader has its own configuration set up with its own log4j.xml file.

Example 10.9. A RepositorySelector

The following code shows a RepositorySelector that looks for a log4j.xml file in the WEB-INF directory.

```
 /*
  * JBoss, Home of Professional Open Source
  * Copyright 2005, JBoss Inc., and individual contributors as indicated
  * by the @authors tag. See the copyright.txt in the distribution for a
  * full listing of individual contributors.
  *
  * This is free software; you can redistribute it and/or modify it
  * under the terms of the GNU Lesser General Public License as
  * published by the Free Software Foundation; either version 2.1 of
  * the License, or (at your option) any later version.
  *
  * This software is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  * Lesser General Public License for more details.
  *
  * You should have received a copy of the GNU Lesser General Public
  * License along with this software; if not, write to the Free
  * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
  * 02110-1301 USA, or see the FSF site: http://www.fsf.org.
  */
package org.jboss.repositoryselectorexample;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.xml.parsers.DocumentBuilderFactory;
import org.apache.log4j.Hierarchy;
import org.apache.log4j.Level;
import org.apache.log4j.LogManager;
import org.apache.log4j.spi.LoggerRepository;
import org.apache.log4j.spi.RepositorySelector;
import org.apache.log4j.spi.RootCategory;
import org.apache.log4j.xml.DOMConfigurator;
import org.w3c.dom.Document;

/**
 * This RepositorySelector is for use with web applications.
 * It assumes that your log4j.xml file is in the WEB-INF directory.
 * @author  Stan Silvert
 */
public class MyRepositorySelector implements RepositorySelector
{
   private static boolean initialized = false;

   // This object is used for the guard because it doesn't get
   // recycled when the application is redeployed.
   private static Object guard = LogManager.getRootLogger();

   private static Map repositories = new HashMap();
   private static LoggerRepository defaultRepository;

   /**
    * Register your web-app with this repository selector.
    */
```

```
  public static synchronized void init(ServletConfig config)
      throws ServletException {
    if( !initialized ) // set the global RepositorySelector
    {
      defaultRepository = LogManager.getLoggerRepository();
      RepositorySelector theSelector = new MyRepositorySelector();
      LogManager.setRepositorySelector(theSelector, guard);
      initialized = true;
    }

    Hierarchy hierarchy = new Hierarchy(new
                          RootCategory(Level.DEBUG));
    loadLog4JConfig(config, hierarchy);
    ClassLoader loader =
        Thread.currentThread().getContextClassLoader();
    repositories.put(loader, hierarchy);
  }

  // load log4j.xml from WEB-INF
  private static void loadLog4JConfig(ServletConfig config,
                                Hierarchy hierarchy)
                                    throws ServletException {
     try {
        String log4jFile = "/WEB-INF/log4j.xml";
        InputStream log4JConfig =

        config.getServletContext().getResourceAsStream(log4jFile);
        Document doc = DocumentBuilderFactory.newInstance()
                                    .newDocumentBuilder()
                                    .parse(log4JConfig);
        DOMConfigurator conf = new DOMConfigurator();
        conf.doConfigure(doc.getDocumentElement(), hierarchy);
     } catch (Exception e) {
        throw new ServletException(e);
     }
   }

  private MyRepositorySelector() {
  }

  public LoggerRepository getLoggerRepository() {
    ClassLoader loader =
         Thread.currentThread().getContextClassLoader();
    LoggerRepository repository =
        (LoggerRepository)repositories.get(loader);

    if (repository == null) {
       return defaultRepository;
    } else {
       return repository;
    }
  }
}
```

10.4. JDK java.util.logging

The choice of the actual logging implementation is determined by the org.jboss.logging.Logger.pluginClass system property. This property specifies the class name of an implementation of the org.jboss.logging.LoggerPlugin interface. The default value for this is the org.jboss.logging.Log4jLoggerPlugin class.

If you want to use the JDK 1.4+ java.util.logging framework instead of log4j, you can create your own Log4jLoggerPlugin to do this. The attached JDK14LoggerPlugin.java file shows an example implementation.

To use this, specify the following system properties:

1. To specify the custom JDK1.4 plugin:

```
org.jboss.logging.Logger.pluginClass = logging.JDK14LoggerPlugin
```

2. To specify the JDK1.4 logging configuration file:

```
java.util.logging.config.file = logging.properties
```

This can be done using the JAVA_OPTS env variable, for example:

```
JAVA_OPTS="-Dorg.jboss.logging.Logger.pluginClass=logging.JDK14LoggerPlugin
        -Djava.util.logging.config.file=logging.properties"
```

You need to make your custom Log4jLoggerPlugin available to JBoss by placing it in a jar in the JBOSS_DIST/lib directory, and then telling JBoss to load this as part of the bootstrap libraries by passing in -L jarname on the command line as follows:

```
starksm@banshee9100 bin$ run.sh -c minimal -L logger.jar
```