Integrating Jakarta Commons Logging with IBM WebSphere Application Server V5

D. A. Zavala
Y. C. Lau

Intended audience and scope

WebSphere J2EE application developers have indicated a need to utilize Jakarta Commons-Logging (JCL) support beyond that supplied by WebSphere Application Server. Typical scenarios include utilizing the Log4J or JDK1.4 logging implementations provided by JCL, utilizing proprietary JCL logger implementations, and incorporating different versions of JCL. This article discusses developing applications and configuring the WebSphere runtime environment towards utilizing application-specific JCL artifacts. We assume the reader is familiar with or has access to references concerning WebSphere Application Server [1,2], J2EE application development [3], and the aforementioned logging facilities [4,5,6,7].

Quick start

Visit section "Building application-specific JCL solutions" to learn procedures for integrating JCL support into WebSphere J2EE applications.

Contents

Acknowledgements

1. Tom Musta, IBM Corporation
2. Don Bourne, IBM Corporation
3. Richard Sitze, IBM Corporation

WebSphere v5 and Jakarta Commons-Logging

Jakarta Commons-Logging (JCL) defines a common programming model for logging and a framework that enables applications to bind different logger implementations; it provides the application programming interfaces (APIs), the NoOpLog and SimpleLog implementations, plus thin "wrapper" implementations over the Apache Log4J, J2SE1.4 and Avalon loggers.

The JCL architecture affords extension by providing a configurable abstract factory facility, LogFactory, to instantiate specified implementations of the LogFactory class. To bind a particular type of Log implementation (i.e. logger) an application configures the LogFactory with the name of the concrete LogFactory implementation class that supports the particular Log implementation. The application invokes method LogFactory.getFactory() to obtain an instance of this class, say *myLogFactory*, and then calls *myLogFactory*.getLog() to obtain the log implementation.

WebSphere Application Server V5 (WAS5) supplies the JCL v1.2 API package, which is a proper subset of the JCL implementation package offered by Apache/Jakarta. WebSphere also supplies proprietary JCL extensions that require the JCL API package to operate. Both artifacts are listed in Appendix A1. WebSphere J2EE applications that utilize JCL will typically obtain an instance of TrLogFactory upon invoking the LogFactory.getFactory() method, and subsequent invocations of TrLogFactory.getLog() will return instances of the WebSphere logging implementation, TrLog.

Applications occasionally must override the WebSphere TrLog logging implementation to bind another, such as the Log4J or JDK1.4 loggers. Further, these applications may require only the classes of the standard JCL package supplied with WebSphere, or may require application-specific JCL artifacts, such as proprietary JCL extensions, different JCL versions, or more comprehensive packages of JCL.

Developers typically realize the requirements above by bundling additional JCL artifacts with their applications and employing options 1, 2, or 3 for JCL LogFactory discovery, as described below. And typically, they encounter unexpected behaviors such as class loading errors or JCL solutions that are mysteriously ineffectual.

Unexpected behaviors concerning JCL integration are due to misunderstanding the JCL LogFactory discovery algorithm and the deployment of application artifacts within the WebSphere runtime environment.

JCL LogFactory specification options and discovery

According to the JCL User's Guide [4], there are three options for specifying the name of the LogFactory implementation class:

1. Assign the fully-qualified name of the LogFactory implementation class to system property org.apache.commons.logging.LogFactory.

   ***Do not use option 1** - Setting a JVM system property is ill advised as it affects all applications running in the JVM, including the application server.*

2. Specify the fully-qualified name of the LogFactory implementation class in file org.apache.commons.logging.LogFactory within the …/META-INF/services directory.

3. Specify a name-value pair assigning the fully-qualified name of the LogFactory implementation class to property org.apache.commons.logging.LogFactory in file commons-logging.properties, and place this file in the classpath[1].

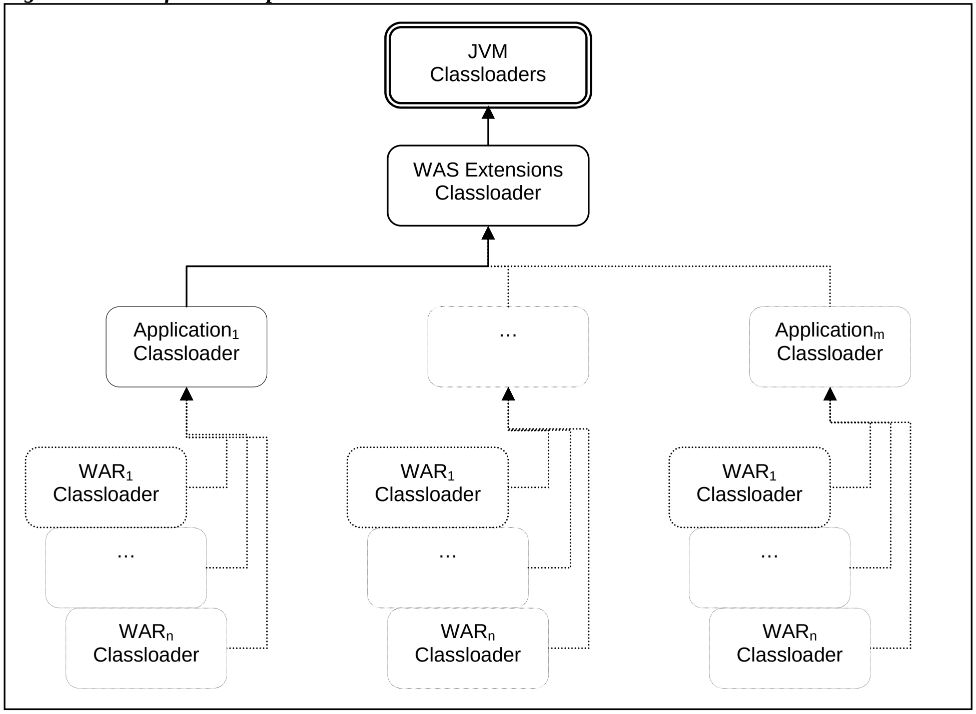
When an application invokes LogFactory.getFactory(), the method performs a greedy algorithm which searches the 3 options above, in the order presented, to obtain the name of the LogFactory implementation class it will attempt to instantiate and return (see Listing 1 in Appendix A2.) Immediately upon finding an item that specifies a LogFactory implementation class that can be instantiated, the search returns an instance of that class. If the search fails, getFactory() returns the default JCL LogFactory implementation, LogFactoryImpl.

Developers can use the files in options 2 and 3 to specify an application-specific LogFactory implementation class. Results of the discovery algorithm will vary depending where the files deploy into the WebSphere runtime (classloader) environment, and how the environment is configured.

The WebSphere classloader environment

By default, applications execute within the WebSphere classloader environment depicted in figure 1.

Figure 1 - The default WebSphere classloader environment



The [delegation] mode for each classloader is *PARENT_FIRST*, the Application classloader policy is *MULTIPLE*, and the WAR classloader policy is *MODULE*. Under the default configuration a unique Application classloader instance exists for each application EAR. All EJB module artifacts, “utility” JARs,

---

[1] The classpath mentioned here will typically be the local classpath of the WebSphere Application or WAR classloader and will depend on how the JCL artifacts are packaged within an application.

and resources bundled with the EAR appear in the local classpath of its corresponding Application classloader. A unique WAR classloader instance exists for each WAR module in the EAR. WAR module artifacts appear in local classpath of its corresponding WAR classloader, each of which is an immediate child of the Application classloader. For describing development procedures we make the following assumption:

> *The policies and modes of the WebSphere classloaders are configured to their defaults until specified otherwise.*

JCL artifacts supplied with WebSphere appear on the local classpath of the WebSphere Extensions classloader[2], which is the parent of all Application classloaders. When an application requires a JCL class or resource in the classloader environment, the context classloader[3] delegates the load operation to its parent classloader. This process recurses upward until reaching the root of the hierarchy – the JVM classloader -- and searches its local classpath for the item. If the search fails, the process "unwinds" to the next lower classloader and searches its local classpath. The process executes until the item is found, or until the search of the context classloader's local classpath fails. In the default classloader environment, JCL classes and resources will typically load from the local classpath of the WebSphere Extensions classloader.

Therein lies the problem. If an application intends to exclusively utilize the JCL artifacts supplied with WebSphere, the scenario above is correct; if an application intends to utilize application-specific JCL artifacts, then the scenario should never occur -- that is, the WebSphere Extensions classloader should never load an application-specific JCL class or resource. The assumption below allows us to safely differentiate the two cases.

> *Applications are fully dependent on either WebSphere JCL artifacts or their own. Application-supplied JCL classes have no dependencies on the JCL classes and resources supplied by WebSphere.*

A general technique to integrate application-specific JCL solutions is to exercise option 2 or 3 to specify the application-specific LogFactory; develop JCL artifacts having no dependencies on JCL classes supplied by WebSphere; deploy these artifacts on the local classpath of a classloader below the WebSphere Extensions classloader; then configure the [delegation] mode of the artifact classloader to PARENT_LAST to ensure it loads application-specific JCL artifacts from its local classpath before delegating load operations upward.

Section "Building application-specific JCL solutions" describes variations on the general technique above. The discussion offers suggestions towards developing JCL artifacts and illustrates procedures for deploying JCL artifacts in application EAR (WAR) and shared libraries, complete with directions for configuring the classloader environment.

In the special case where an application requires only the standard JCL support supplied with WebSphere, visit section "Using WebSphere JCL artifacts."

Building application-specific JCL solutions

Applications frequently require JCL support beyond that supplied by WebSphere -- for example, a Web application that utilizes the JCL Log4J logger. JCL Log4J support is not in the JCL API package supplied by WebSphere, but can be obtained from Apache/Jakarta. Applications may require proprietary JCL extensions as well.

---

[2] Section 17.6 of the *IBM WebSphere Application Server V5.0 System Management and Configuration* Redbook [1] provides more details about of the WebSphere classloader structure.

[3] Just a reminder, the "context classloader" is initially the Application classloader on EJB method invocations, and the WAR classloader on Servlet/JSP service() method invocations.

In general, to build application-specific JCL solutions:

1. Specify the desired LogFactory implementation within one of the files below.
   - org.apache commons.logging.LogFactory (JCL option 2)
   - commons-logging.properties. (JCL option 3)
2. Package all JCL classes and resources in a utility JAR, say *commons-logging.jar*, including the file created in step 1.
3. Make *commons-logging.jar* available to the application using one of these techniques:
   - Add *commons-logging.jar* to the application EAR
   - Add *commons-logging.jar* to the application WAR
   - Configure *commons-logging.jar* into an application-associated shared library.
   - Configure *commons-logging.jar* into a server-associated shared library.
4. Set the [delegation] mode of the WebSphere classloader that contains *commons-logging.jar* in its local classpath to *PARENT_LAST*.

Subsequent sections present development solutions that describe when these options are appropriate and how to successfully execute them.

Regarding the steps above, there is no preference regarding which JCL file to specify the LogFactory implementation class in step 1. Each works equally well given you follow the solutions.

To avoid potential problems we suggest in step 2 packaging all application-specific JCL classes and resources in a single utility JAR. Also, we assume application-specific JCL classes and resources support the application independently of those supplied by WebSphere. These two conditions eliminate potential versioning issues between the application- and WebSphere-supplied JCL classes. Consider the case where an application-specific JCL class is dependent on a class supplied by WebSphere, and the two classes are incompatible. The conditions also ensure the same classloader will load all resources of the JCL solution -- the classloader containing this JAR on its local classpath – which facilitates debugging in the event a classloader anomaly occurs, such as a JCL class is not found or a class cast exception.

Adding application-specific JCL artifacts to EARs (WARs) using utility JARs is a preferred practice, because the artifacts are implicitly versioned per application, and because they are automatically deployed to the server environment during application installation. At runtime the JARs are added to the local classpath of the specific WebSphere Application (WAR) classloader.

Shared libraries provide a means to introduce application artifacts into the WebSphere runtime environment without adding these artifacts to EARs, EJB JARs, or WARs. Shared libraries are managed via the Admin console or wsadmin scripts. Once a shared library is defined, it may be associated to an application or to a user-defined “server” classloader. The former is called an “application-associated” shared library; the latter is “server-associated.”

Configuring application artifacts into shared libraries has immediate benefits. Shared libraries containing different JCL solutions, for instance, can be associated with an application without uninstalling and reconfiguring the application. Alternately, different applications can be associated with the same JCL solution, facilitating system management with a single point of maintenance rather than maintaining various JCL artifacts packaged within each application. By associating a shared library containing JCL artifacts to a “server” classloader, a single JCL solution is available to all applications hosted by a particular server, without any configuration to the applications. Thus, employing shared libraries can facilitate application versioning, maintenance, and development.

Finally, setting classloader delegation mode to *PARENT_LAST* is always necessary to ensure the appropriate WebSphere classloader loads all the classes and resources of the JCL solution before delegating the load operation to its parent, potentially causing integration problems.

Here are some suggested approaches to building application-specific JCL solutions.

Solution - JCL option 3 + EAR

Use this approach if your application contains only EJBs, or EJBs and Servlets/JSPs that require JCL support.

To provide an application-specific JCL solution using JCL option 3, the commons-logging.properties file, in an Enterprise Application Archive (EAR):

1. Specify the desired LogFactory implementation within file commons-logging.properties:

    a. Create a file named “commons-logging.properties”
    b. Enter the following line into the file:

    org.apache.commons.logging.LogFactory=<fully_qualified_LogFactory_implementation_class>

    For example, to use the Log4J factory, enter the line shown in figure 2:

Figure 2 - Content of file commons-logging.properties

```
org.apache.commons.logging.LogFactory=org.apache.commons.logging.impl.Log4jFactory
```

2. Package all JCL classes and resources in a utility JAR, say *commons-logging.jar*, inserting the commons-logging.properties file into the root directory of that JAR [figure 3].

Figure 3 - Partial listing, commons-logging.jar file

```
META-INF
META-INF/LICENSE.txt
META-INF/MANIFEST.MF
commons-logging.properties
org
org/apache
org/apache/commons
org/apache/commons/logging
org/apache/commons/logging/Log.class
…
```

The general requirement for the commons-logging.properties file to be effective is that the file be visible on the application classpath. In a multi-classloader environment, this requirement is not straightforward, since various classloaders may load application artifacts. For simplicity, we suggest inserting the commons-logging.properties file into the JAR containing application-specific JCL artifacts. This satisfies the requirement via a standard J2EE application development practice.

3. Make *commons-logging.jar* available to EJB and Web modules of your application.

    a. Add file *commons-logging.jar* to the root directory of the application EAR [figure 4].

Figure 4 - Partial listing of application EAR containing JCL artifact

```
META-INF
META-INF/MANIFEST.MF
...
commons-logging.jar
...
```

b. Add *commons-logging.jar* to the Class-Path attribute within the MANIFEST.MF file of every EJB JAR and WAR dependent on the JCL solution. At minimum, the Class-Path attribute should appear as shown in figure 5.

Figure 5 - Class-Path attribute of EJB module META-INF/MANIFEST.MF

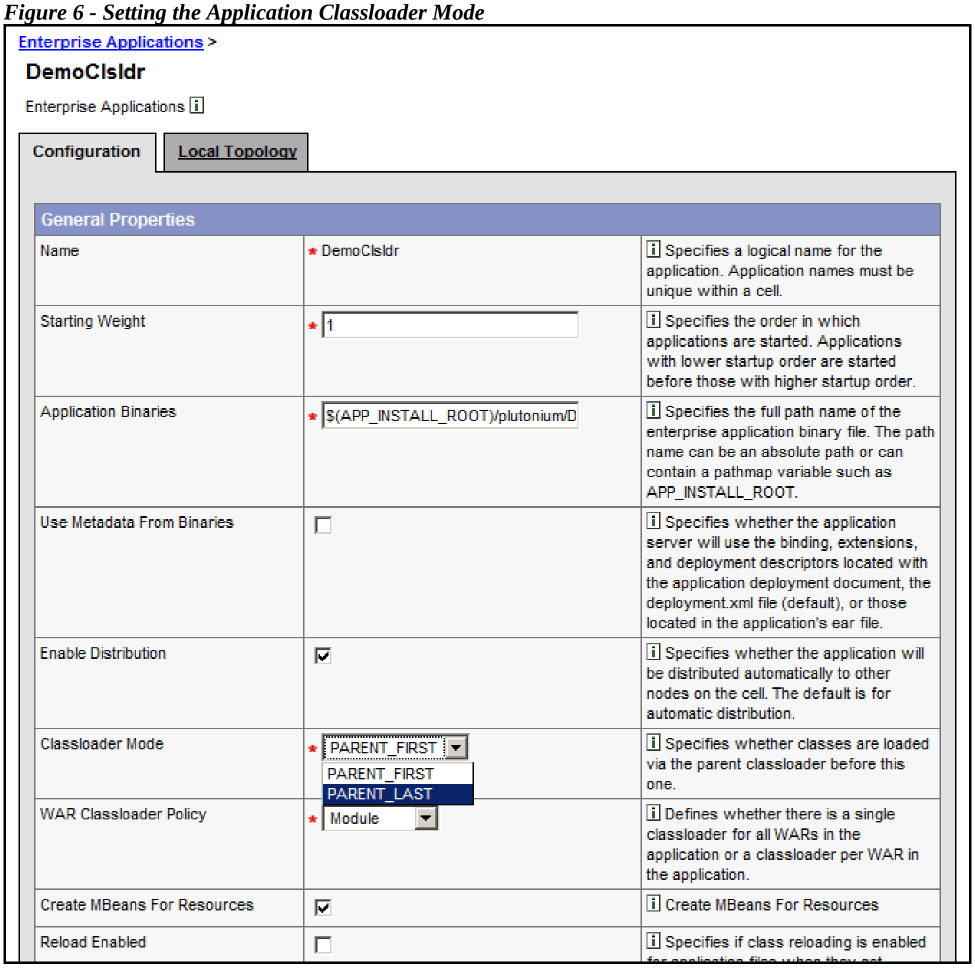
```
Class-Path: commons-logging.jar
```

At runtime *commons-logging.jar* will appear on the local classpath of the Application classloader. For more information, visit section 17.7.1 of System Management and Configuration Redbook [1] for directions to package utility JARs into WARs and EARs.

4. Set the [delegation] mode of the Application classloader to PARENT_LAST. Using the Admin console:

   a. Select Applications > Enterprise Applications > “my application” [figure 6.]
   b. Select PARENT_LAST from the Classloader Mode drop-down list.
   c. Click OK.
   d. Select Save and save your changes!

Figure 6 - Setting the Application Classloader Mode



Enterprise Applications >

DemoClsldr

Enterprise Applications

General Properties		
Name	* DemoClsldr	Specifies a logical name for the application. Application names must be unique within a cell.
Starting Weight	* 1	Specifies the order in which applications are started. Applications with lower startup order are started before those with higher startup order.
Application Binaries	* $(APP_INSTALL_ROOT)/plutonium/D	Specifies the full path name of the enterprise application binary file. The path name can be an absolute path or can contain a pathmap variable such as APP_INSTALL_ROOT.
Use Metadata From Binaries	☐	Specifies whether the application server will use the binding, extensions, and deployment descriptors located with the application deployment document, the deployment.xml file (default), or those located in the application's ear file.
Enable Distribution	☑	Specifies whether the application will be distributed automatically to other nodes on the cell. The default is for automatic distribution.
Classloader Mode	* PARENT_FIRST PARENT_FIRST PARENT_LAST	Specifies whether classes are loaded via the parent classloader before this one.
WAR Classloader Policy	* Module	Defines whether there is a single classloader for all WARs in the application or a classloader per WAR in the application.
Create MBeans For Resources	☑	Create MBeans For Resources
Reload Enabled	☐	Specifies if class reloading is enabled

5. Restart the application server for the changes to take effect.

The server configuration changes are now effective, allowing the JCL solution to function as expected.

Solution - JCL option 3 + WAR

Use this approach if your application contains only Servlets/JSPs that require JCL support.

To build an application-specific JCL solution employing JCL option 3, the commons-logging.properties file, in a Web module (WAR):

1. Specify the desired LogFactory implementation within file commons-logging.properties -- see step 1 of section "JCL option 3 + EAR"

2. Package all JCL classes and resources in a utility JAR, say *commons-logging.jar*, inserting the commons-logging.properties file into the root directory of that JAR -- see step 2 of section “JCL option 3 + EAR”

3. Make *commons-logging.jar* available to the Web modules of your application.

   Add file *commons-logging.jar* to the /WEB-INF/lib directory of each application WAR that requires JCL support [figure 7.]

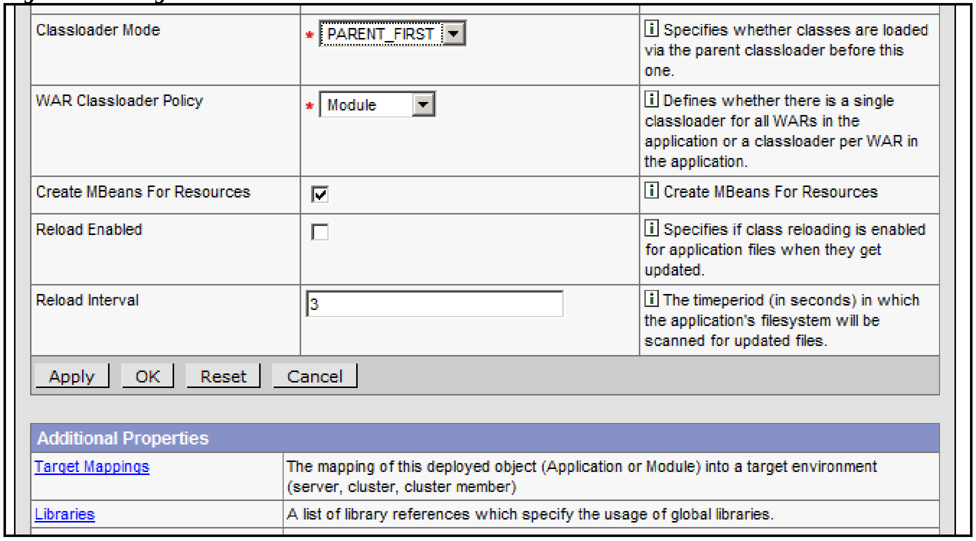
Figure 7 - Partial listing of application WAR containing JCL utility jar

```
...
WEB-INF/lib/commons-logging.jar
...
```

4. Set the [delegation] mode of the application’s WAR classloader to PARENT_LAST. Using the Admin console:

   a. Select Applications > Enterprise Applications > “my application” > Web Modules > “my WAR module” [figure 8]
   b. Select PARENT_LAST from the Classloader Mode drop-down list.
   c. Select Save and save your changes!

Figure 8 - Setting the WAR Classloader Mode



Classloader Mode	* PARENT_FIRST	Specifies whether classes are loaded via the parent classloader before this one.
WAR Classloader Policy	* Module	Defines whether there is a single classloader for all WARs in the application or a classloader per WAR in the application.
Create MBeans For Resources	☑	Create MBeans For Resources
Reload Enabled	☐	Specifies if class reloading is enabled for application files when they get updated.
Reload Interval	3	The timeperiod (in seconds) in which the application's filesystem will be scanned for updated files.

Additional Properties	
Target Mappings	The mapping of this deployed object (Application or Module) into a target environment (server, cluster, cluster member)
Libraries	A list of library references which specify the usage of global libraries.

5. Restart the application server.

The server configuration changes are now effective, allowing the JCL solution to function as expected.

Solution - JCL option 2 + application-associated shared library

Use this approach if your application contains only EJBs, or EJBs and Servlets/JSPs that require JCL support, and you would like to utilize this solution across select applications by modifying the server configuration rather than just the application (EAR.)

To build an application-specific JCL solution employing JCL option 2, the org.apache.commons.logging.LogFactory file, in an application-associated shared library:

1. Specify the desired LogFactory implementation within file org.apache.commons.logging.LogFactory.

    a. Create a file named “org.apache.commons.logging.LogFactory”.
    b. Insert into it the name of the JCL LogFactory implementation class, without quotes!

    Figure 9 lists the contents of an org.apache.commons.logging.LogFactory file which specifies the default JCL LogFactory class, org.apache.commons.logging.impl.LogFactoryImpl.

Figure 9 - The org.apache.commons.logging.LogFactory file

```
org.apache.commons.logging.impl.LogFactoryImpl
```

2. Package all JCL classes and resources in a utility JAR, say *commons-logging.jar*, inserting the org.apache.commons.logging.LogFactory file into the /META-INF/services directory of that JAR.

    Upon completing step 2, a listing of the utility JAR should resemble figure 10. Ensure that the LogFactory file resides in the META-INF/services directory, which in turn resides at the root level of the JAR, and that the names are correct.

Figure 10 - Partial listing, commons-logging.jar file

```
META-INF
META-INF/LICENSE.txt
META-INF/MANIFEST.MF
META-INF/services/ org.apache.commons.logging.LogFactory
org
org/apache
org/apache/commons
org/apache/commons/logging
org/apache/commons/logging/Log.class
…
```

3. Make *commons-logging.jar* available to the EJB and Web modules of your application by configuring it into shared library and associating that library to your application.

    a. Install your J2EE application. Do not start the application yet!
    b. Copy *commons-logging.jar* to a directory that is preferably outside the WAS install. For sake of example we use “d:\tmp\jcl\”.

    Create the shared library. Shared libraries can be created using the Admin console or wsadmin scripting. Using the Admin console:

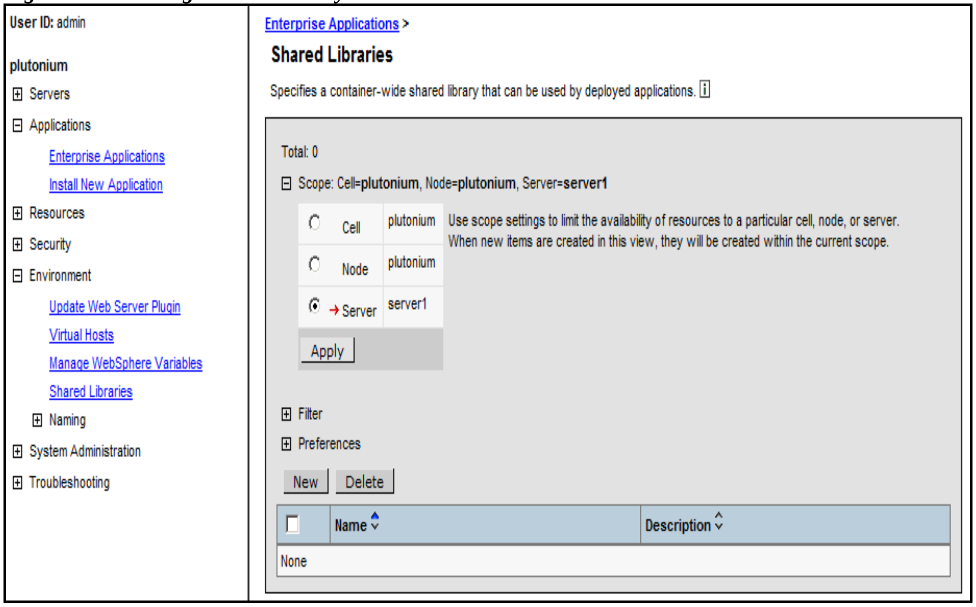
    d. Click “Environment” > “Shared Libraries”. You should see the screen in figure 11.
    e. Select scope “➔Server”.
    f. Click “New”.
    g. Enter the name, description, and classpaths for the shared library [figure 12.]

   i. Enter “JCL” in the Name field.
   ii. [Optional] Enter “JCL shared library” in the Description field.
   iii. Enter “d:\tmp\jcl\commons-logging.jar” in the Classpath field.

h. Click OK.
i. Select Save and save your changes.

The shared library is now defined within the WebSphere server configuration.

Figure 11 - Creating a shared library



Shared Libraries

Specifies a container-wide shared library that can be used by deployed applications.

Total: 0

⊟ Scope: Cell=**plutonium**, Node=**plutonium**, Server=**server1**

○ Cell	plutonium	Use scope settings to limit the availability of resources to a particular cell, node, or server. When new items are created in this view, they will be created within the current scope.
○ Node	plutonium	
● Server	server1	

☐	Name	Description
None		

Associate the shared library to the application that requires the JCL solution. Again this can be achieved using the Admin console or wsadmin scripting. Using the Admin console:

j. Select Applications > Enterprise Applications > “my application”
k. Select Libraries towards the bottom of the screen. The screen in figure 13 appears.
l. Click Add.
m. Select the name of the shared library containing the JCL solution from the Library Name drop down list [figure 14.]
n. Click OK

The shared library is now bound to the application.

4. Set the Application classloader [delegation] mode to “PARENT_LAST”. Using the Admin console:

   a. Select Applications > Enterprise Applications > “my application”.
   b. Select PARENT_LAST from the Classloader Mode drop-down list as shown in figure 6.
   c. Click OK.
   d. Select Save and save your changes.

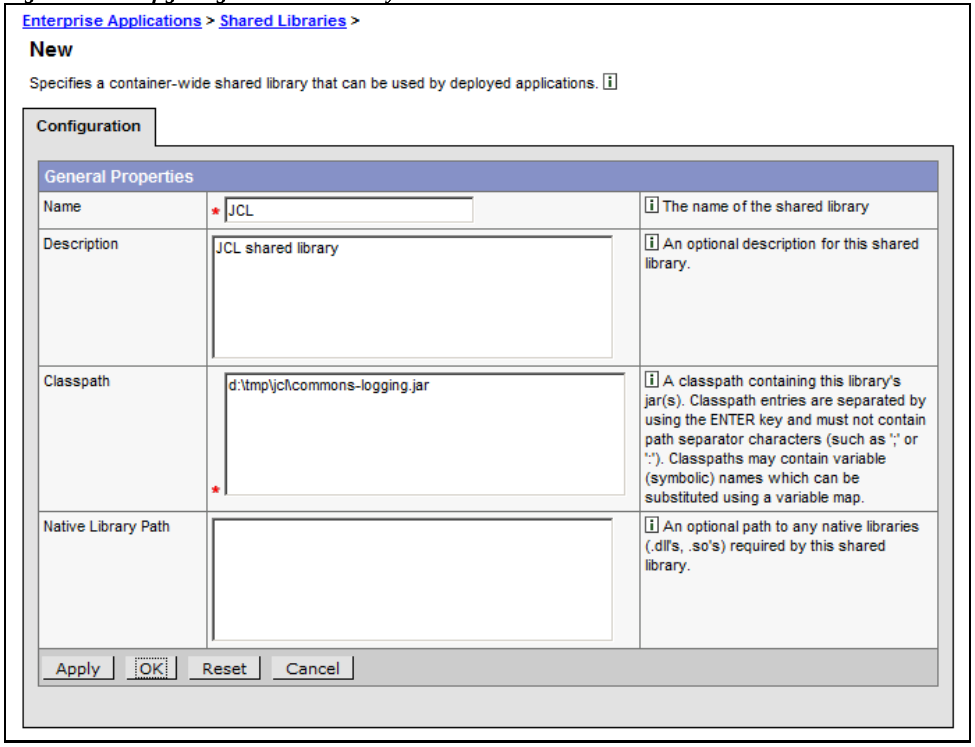
The mode of the Application classloader is now PARENT_LAST.

5. Restart the server for the changes to take effect.

The JCL solution is now accessible to the application. The application server adds the shared library's classpath to the local classpath of the Application classloader only, which causes the application's EJB and Web modules to utilize the JCL solution contained in the shared library.

For more information about configuring shared libraries, visit the IBM WebSphere Application Server V5 InfoCenter [2].

Figure 12 - Configuring the shared library



Enterprise Applications > Shared Libraries >

New

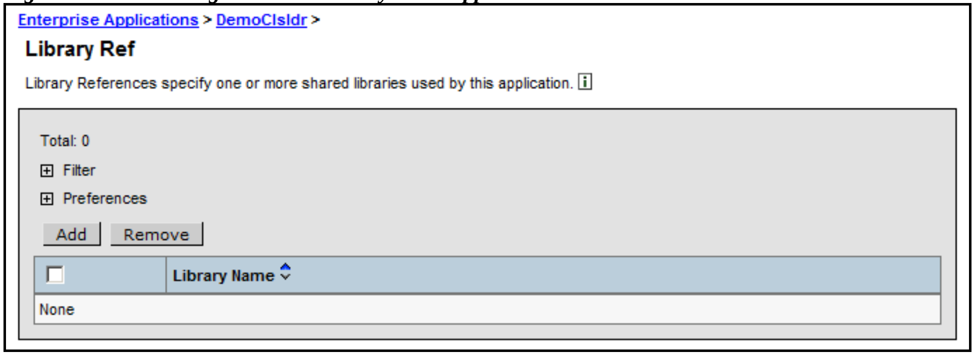
Specifies a container-wide shared library that can be used by deployed applications.

Configuration

General Properties

Name	* JCL	The name of the shared library
Description	JCL shared library	An optional description for this shared library.
Classpath	* d:\tmp\jcl\commons-logging.jar	A classpath containing this library's jar(s). Classpath entries are separated by using the ENTER key and must not contain path separator characters (such as ';' or ':'). Classpaths may contain variable (symbolic) names which can be substituted using a variable map.
Native Library Path		An optional path to any native libraries (.dll's, .so's) required by this shared library.

Figure 13 - Associating the shared library to the application



Enterprise Applications > DemoClsldr >

Library Ref

Library References specify one or more shared libraries used by this application.

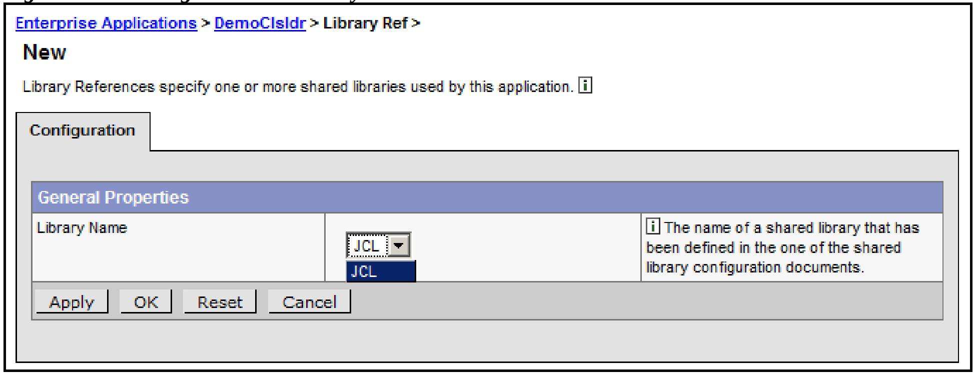
Total: 0

⊞ Filter

⊞ Preferences

☐	Library Name
None	

Figure14 - Selecting the shared library



Enterprise Applications > DemoClsldr > Library Ref >

New

Library References specify one or more shared libraries used by this application.

Library Name: JCL

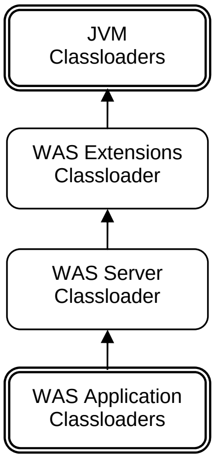
The name of a shared library that has been defined in the one of the shared library configuration documents.

Solution - JCL option 3 + server-associated shared library

Use this approach if your application contains only EJBs, or EJBs and Servlets/JSPs that require JCL support, and you would like to utilize this solution across all applications hosted by an application server.

Deploying application artifacts in server-associated shared library does not involve the Application or WAR classloaders, but rather, introduces a user-defined "server" classloader between the WebSphere Extensions classloader and the Application classloaders as depicted below. Adding a server-associated shared library containing JCL artifacts will make the JCL solution visible to all applications hosted by the server which contains the user-defined classloader. For this reason the solution may be desirable in special development scenarios.

Figure 15 - The WebSphere classloader environment with a user-defined classloader



To build an application-specific JCL solution employing JCL option 3, the commons-logging.properties file, in a server-associated shared library:

1. Specify the desired LogFactory implementation in the commons-logging.properties file -- see step 1 of section “JCL option 3 + EAR”.

2. Package all JCL classes and resources in a utility JAR, say *commons-logging.jar*, inserting the commons-logging.properties file into the root directory of that JAR -- see step 2 of section “JCL option 3 + EAR”.

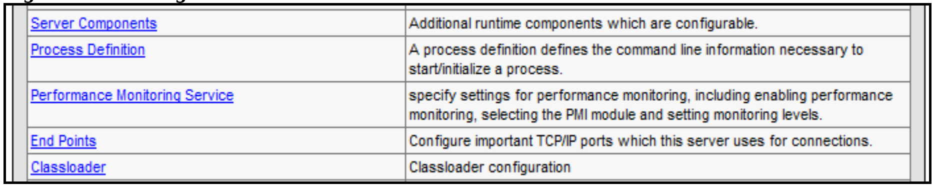
3. Make *commons-logging.jar* available to the EJB and Web modules of all applications by configuring it into shared library and associating that library to a user-defined server classloader.

    a. Follow steps 3.a through 3.i presented in section "Solution - JCL option 3 + application-associated shared library" to install the application and define a shared library.

    Create a user-defined “server” classloader configured with PARENT_LAST delegation. This may be achieved using the Admin console or wsadmin scripting. Using the Admin console:

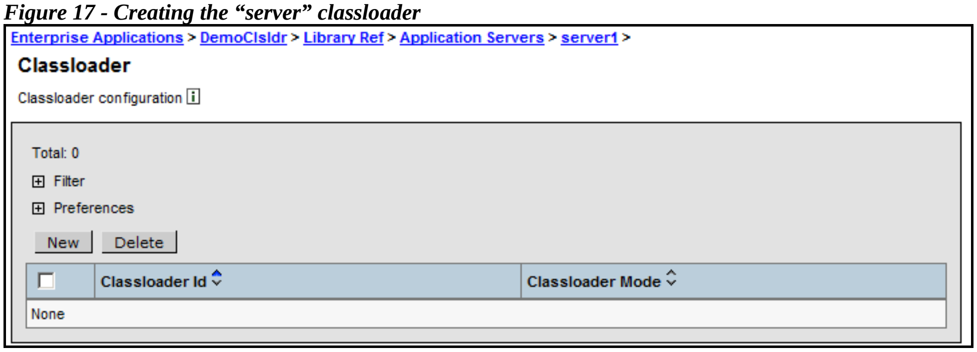
    b. Select Servers > “my server” > Classloader. The screen in figure 16 shows the Classloader link; the screen in figure 17 appears.
    c. Click New.
    d. Select PARENT_LAST from the Classloader Mode drop-down menu [figure 18.]
    e. Click OK.
    f. Select Save and save your change.

Figure 16 - Finding the “server” classloader link



Server Components	Additional runtime components which are configurable.
Process Definition	A process definition defines the command line information necessary to start/initialize a process.
Performance Monitoring Service	specify settings for performance monitoring, including enabling performance monitoring, selecting the PMI module and setting monitoring levels.
End Points	Configure important TCP/IP ports which this server uses for connections.
Classloader	Classloader configuration

Figure 17 - Creating the “server” classloader



Enterprise Applications > DemoClsldr > Library Ref > Application Servers > server1 >

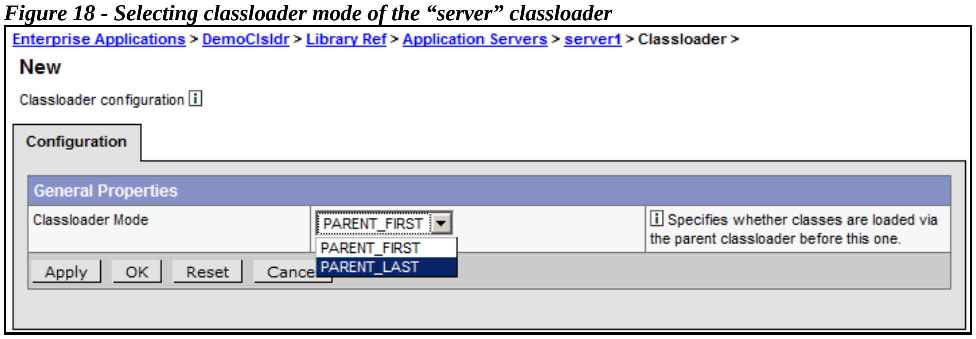
Classloader

Classloader configuration

Total: 0

☐	Classloader Id	Classloader Mode
None		

Figure 18 - Selecting classloader mode of the "server" classloader



Enterprise Applications > DemoClsldr > Library Ref > Application Servers > server1 > Classloader >

New

Classloader configuration

Configuration

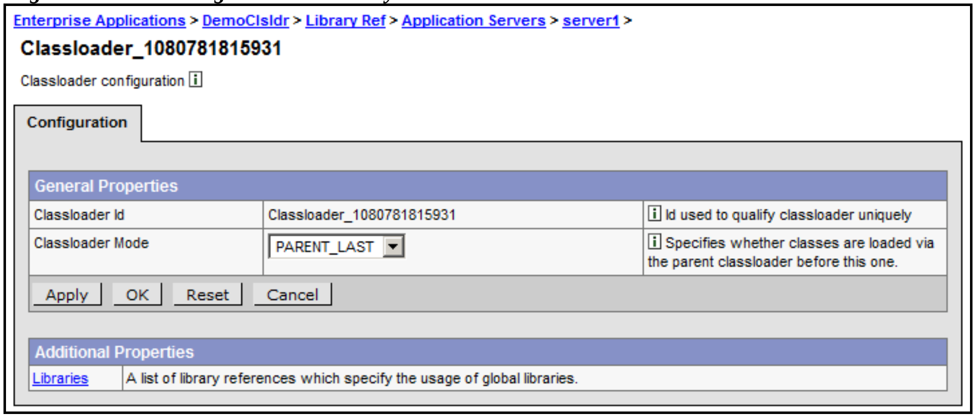
General Properties

Classloader Mode: PARENT_FIRST — Specifies whether classes are loaded via the parent classloader before this one.

4. Associate the shared library to the classloader. Again, this can be achieved using the Admin console or wsadmin scripting. Using the Admin console:

   a. Select Servers > "my server" > Classloader.
   b. Select the classloader created above. The screen in figure 19 appears.
   c. Select Libraries.
   d. Click Add.
   e. Select the shared library created in step 3 from the Library Name drop-down menu.
   f. Click OK.
   g. Select Save and save the changes.

Figure 19 - Associating the shared library to the "server" classloader



Enterprise Applications > DemoClsldr > Library Ref > Application Servers > server1 >

Classloader_1080781815931

Classloader configuration

Configuration

General Properties

Classloader Id	Classloader_1080781815931	Id used to qualify classloader uniquely
Classloader Mode	PARENT_LAST	Specifies whether classes are loaded via the parent classloader before this one.

Additional Properties

Libraries	A list of library references which specify the usage of global libraries.

6. Restart the server for the changes to take effect.

The JCL solution is now accessible to all applications hosted by the server. The application server adds the shared library's classpath to the local classpath of the specified "server" classloader only, which causes the all applications hosted by the server to potentially utilize the JCL solution contained in the shared library. For more information about configuring shared libraries, visit the IBM WebSphere Application Server V5 InfoCenter [2].

Using WebSphere JCL artifacts

An application may require only the standard JCL classes and resources supplied by WebSphere [Appendix A1.1.] Although use of these artifacts is not officially supported, developers have attempted to use them. So let's examine this special case. The simplest way to use WebSphere JCL support is to specify the LogFactory implementation class in file org.apache.commons.logging.LogFactory (see option 2 in section "JCL LogFactory specification and discovery.") Typically, such applications obtain the default JCL LogFactory. There is no need for *PARENT_LAST* delegation, because there are no application-specific JCL artifacts to deploy, and because the solution does not utilize file commons-logging.properties (see option 3, section "JCL LogFactory specification and discovery.")

Solution - JCL option 2

To utilize JCL classes supplied by WebSphere:

1. Specify the desired LogFactory implementation in the LogFactory file.

    a. Create a file named "org.apache.commons.logging.LogFactory".
    b. Insert into it the name of the JCL LogFactory implementation class, without quotes!

Figure 9 lists the contents of an org.apache.commons.logging.LogFactory file which specifies the default JCL LogFactory class, org.apache.commons.logging.impl.LogFactoryImpl.

2. Add file org.apache.commons.logging.LogFactory to the META-INF/services directory of the EJB or Web module (i.e., JAR or WAR file) that requires the JCL logger.

Adding the LogFactory file to an EJB JAR ensures the Application classloader loads the file; adding the file to a WAR does the same for the WAR classloader. The EJB-solution will apply to all EJB and Web modules; the WAR-solution applies to Web modules only.

Upon completing step 2, a listing of the EJB JAR or WAR contents should be similar to figure 22. Ensure that the LogFactory files reside in the META-INF/services directory, which in turn resides at the root level of the EJB or WAR module, and that the names are correct.

Figure 20 - A partial WAR listing containing the org.apache.commons.logging.LogFactory file

```
META-INF/
META-INF/MANIFEST.MF
META-INF/services/
META-INF/services/org.apache.commons.logging.LogFactory
...
```

Why this works

Adding the org.apache.commons.logging.LogFactory file to the META-INF/services directory of the EJB JAR (WAR) effectively appends the file to local classpath of the WebSphere Application (WAR) classloader, because the Java Extensions/Service Provider facility of the classloaders automatically searches the META-INF/services directory during load operations. When an application invokes LogFactory.getFactory(), the LogFactory discovery algorithm attempts to open the org.apache.commons.logging.LogFactory file. The context classloader searches for the file according to its [delegation] mode. Under *PARENT_FIRST* delegation the classloader delegates the search up the WebSphere classloader tree, but the search ultimately fails, because WebSphere specifies its LogFactory implementation class in file commons-logging.properties located in file ws-commons-logging.jar [Appendix A1.2], not org.apache.commons.logging.LogFactory. And thus, the context classloader finds the file on its own classpath.

If the search for the org.apache.commons.logging.LogFactory file fails, or if the class specified inside the file cannot be instantiated, then the getFactory() method searches for the commons-logging properties file. Under *PARENT_FIRST* delegation, it will find the file provided by WebSphere and thus return an instance of TrLogFactory. Failure to find the org.apache.commons.logging.LogFactory file or instantiate the class named within it will likely be caused by typing errors when constructing the file.

> ***Caveat*** *- If future versions of WebSphere employ the LogFactory file to integrate commons-logging features, this solution will be ineffectual unless PARENT_LAST classloader delegation is configured for the classloader containing the LogFactory file in its local classpath.*

Adding JCL artifacts to the JVM classpath

Finally, adding an application's JCL artifacts to the server JVM classpath *may be ineffectual* as the Java Extensions classloader (the classloader to which this classpath setting applies) resides above the WebSphere Extensions classloader in the WebSphere runtime environment. Never utilize this approach to add application artifacts to the WebSphere runtime environment.

References

1. IBM International Technical Support Organization. *IBM WebSphere Application Server V5.0 System Management and Configuration; WebSphere Handbook Series,* SG24-6195-00. IBM Corporation. April 2003.
   http://www.redbooks.ibm.com/redbooks/pdfs/sg246195.pdf

2. IBM WebSphere V5.0 InfoCenter.
   http://www-306.ibm.com/software/webservers/appserv/infocenter.html.

3. Sun J2EE platform.
   http://java.sun.com/j2ee/index.jsp.

4. Apache-Jakarta Commons Logging Users Guide.
   http://jakarta.apache.org/commons/logging/userguide.html.

5. Apache-Jakarta Commons Logging Overview.
   http://java.sun.com/j2se/1.4.2/docs/guide/logging/overview.html.

6. Sun JDK1.4 Logging API.
   http://java.sun.com/j2se/1.4/docs/guide/util/logging/index.html.

7. Apache-Avalon Toolkit.
   http://avalon.apache.org/logkit/index.html.

8. Sun J2SE JAR File Specification.
   http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html

Appendices

Appendix A1 - WebSphere commons-logging artifacts

WebSphere supplies two JCL artifacts: ws-commons-logging.jar, which contains WebSphere proprietary JCL extensions, and commons-logging-api.jar, which contains the JCL support required by the extensions.

Figure A1.1 - WebSphere-supplied JCL API package, commons-logging-api.jar

```
META-INF/
META-INF/MANIFEST.MF
org/
org/apache/
org/apache/commons/
org/apache/commons/logging/
org/apache/commons/logging/impl/
org/apache/commons/logging/impl/Jdk14Logger.class
org/apache/commons/logging/impl/LogFactoryImpl$1.class
org/apache/commons/logging/impl/LogFactoryImpl.class
org/apache/commons/logging/impl/NoOpLog.class
org/apache/commons/logging/impl/SimpleLog$1.class
org/apache/commons/logging/impl/SimpleLog.class
org/apache/commons/logging/Log.class
org/apache/commons/logging/LogFactory$1.class
org/apache/commons/logging/LogFactory$2.class
org/apache/commons/logging/LogFactory$3.class
org/apache/commons/logging/LogFactory.class
org/apache/commons/logging/LogConfigurationException.class
org/apache/commons/logging/LogSource.class
META-INF/LICENSE.txt
```

Figure A1.2 - WebSphere JCL extensions, ws-commons-logging.jar

```
META-INF/MANIFEST.MF
META-INF/
com/
com/ibm/
com/ibm/ws/
com/ibm/ws/commons/
com/ibm/ws/commons/logging/
com/ibm/ws/commons/logging/TrLog.class
com/ibm/ws/commons/logging/TrLogFactory.class
commons-logging.properties
```

Appendix A2 - JCL LogFactory.getFactory() implementation

Deleted: Page Break

The JCL LogFactory discovery algorithm.

Figure A2.1 - LogFactory.getFactory( ) listing

```
public static LogFactory getFactory()
    throws LogConfigurationException {

  // Identify the classloader we will be using
  ClassLoader contextClassLoader = (ClassLoader)AccessController.doPrivileged(
      new PrivilegedAction() {
        public Object run() {return getContextClassLoader();}
      });

  // Return any previously registered factory for this classloader
  LogFactory factory = getCachedFactory(contextClassLoader);
  if (factory != null)
```

```
    return factory;

  // Load properties file..
  // will be used one way or another in the end.
  Properties props=null;
  try {
    InputStream stream = getResourceAsStream(contextClassLoader,
FACTORY_PROPERTIES);
    if (stream != null) {
      props = new Properties();
      props.load(stream);
      stream.close();
    }
  } catch (IOException e) {
  } catch (SecurityException e) {
  }

  // First, try the system property
  try {
    String factoryClass = System.getProperty(FACTORY_PROPERTY);
    if (factoryClass != null) {
      factory = newFactory(factoryClass, contextClassLoader);
    }
  } catch (SecurityException e) {
    ;  // ignore
  }

  // Second, try to find a service by using the JDK1.3 JAR
  // discovery mechanism. This will allow users to plug a logger
  // by just placing it in the lib/ directory of the webapp ( or in
  // CLASSPATH or equivalent ). This is similar with the second
  // step, except that it uses the (standard?) jdk1.3 location in the JAR.
  if (factory == null) {
    try {
      InputStream is = getResourceAsStream(contextClassLoader, SERVICE_ID);
      if( is != null ) {
        // This code is needed by EBCDIC and other strange systems.
        // It's a fix for bugs reported in xerces
        BufferedReader rd;
        try {
          rd = new BufferedReader(new InputStreamReader(is, "UTF-8"));
        } catch (java.io.UnsupportedEncodingException e) {
          rd = new BufferedReader(new InputStreamReader(is));
           }

        String factoryClassName = rd.readLine();
        rd.close();
        if (factoryClassName != null &&
            ! "".equals(factoryClassName)) {
          factory= newFactory( factoryClassName, contextClassLoader );
        }
      }
    } catch( Exception ex ) {
      ;
    }
  }

  // Third try a properties file.
  // If the properties file exists, it'll be read and the properties
  // used. IMHO ( costin ) System property and JDK1.3 JAR service
  // should be enough for detecting the class name. The properties
  // should be used to set the attributes ( which may be specific to
  // the webapp, even if a default logger is set at JVM level by a
  // system property )

  if (factory == null  &&  props != null) {
    String factoryClass = props.getProperty(FACTORY_PROPERTY);
    if (factoryClass != null) {
      factory = newFactory(factoryClass, contextClassLoader);
    }
```

```
}

// Fourth, try the fallback implementation class
if (factory == null) {
  factory = newFactory(FACTORY_DEFAULT, LogFactory.class.getClassLoader());
}
if (factory != null) {
  /**
   Always cache using context classloader..
   */
  cacheFactory(contextClassLoader, factory);

  if( props!=null ) {
    Enumeration names = props.propertyNames();
    while (names.hasMoreElements()) {
      String name = (String) names.nextElement();
      String value = props.getProperty(name);
      factory.setAttribute(name, value);
            }
      ...
```

Appendix A3 - The JCL implementation package, commons-logging.jar

The commons-logging.jar archive contains the entire set of JCL classes released by Jakarta.

Figure A3.1 - The JCL implementation package, commons-logging.jar

```
META-INF
META-INF/LICENSE.txt
META-INF/MANIFEST.MF
org
org/apache
org/apache/commons
org/apache/commons/logging
org/apache/commons/logging/Log.class
org/apache/commons/logging/LogConfigurationException.class
org/apache/commons/logging/LogFactory$1.class
org/apache/commons/logging/LogFactory$2.class
org/apache/commons/logging/LogFactory$3.class
org/apache/commons/logging/LogFactory.class
org/apache/commons/logging/LogSource.class
org/apache/commons/logging/package.html
org/apache/commons/logging/impl
org/apache/commons/logging/impl/Jdk14Logger.class
org/apache/commons/logging/impl/Log4JCategoryLog.class
org/apache/commons/logging/impl/Log4jFactory.class
org/apache/commons/logging/impl/Log4JLogger.class
org/apache/commons/logging/impl/LogFactoryImpl$1.class
org/apache/commons/logging/impl/LogFactoryImpl.class
org/apache/commons/logging/impl/LogKitLogger.class
org/apache/commons/logging/impl/NoOpLog.class
org/apache/commons/logging/impl/package.html
org/apache/commons/logging/impl/SimpleLog$1.class
org/apache/commons/logging/impl/SimpleLog.class
```