



WebSphere Application Server V7: Understanding Class Loaders

Understanding how the Java™ and WebSphere® class loaders work is critical to packaging and deploying Java EE5 applications. Failure to set up the class loaders properly most likely results in a cascade of the infamous class loading exceptions (such as `ClassNotFoundException`) when trying to start your application.

In this chapter we explain class loaders and how to customize the behavior of the WebSphere class loaders to suit your particular application's requirements. The chapter concludes with an example designed to illustrate these concepts.

We cover the following topics:

- ▶ “A brief introduction to Java class loaders” on page 2
- ▶ “WebSphere class loaders overview” on page 6
- ▶ “Configuring WebSphere for class loaders” on page 10
- ▶ “Class loader viewer” on page 17
- ▶ “Learning class loaders by example” on page 18

A brief introduction to Java class loaders

Class loaders enable the Java virtual machine (JVM™) to load classes. Given the name of a class, the class loader locates the definition of this class. Each Java class must be loaded by a class loader.

When you start a JVM, you use three class loaders: the bootstrap class loader, the extensions class loader, and the application class loader:

- ▶ The *bootstrap* class loader is responsible for loading only the core Java libraries in the *Java_home/jre/lib* directory. This class loader, which is part of the core JVM, is written in native code.
- ▶ The *extensions* class loader is responsible for loading the code in the extensions directories (*Java_home/jre/lib/ext* or any other directory specified by the *java.ext.dirs* system property). This class loader is implemented by the *sun.misc.Launcher\$ExtClassLoader* class.
- ▶ The *application class loader* is responsible for loading code that is found on *java.class.path*, which ultimately maps to the system *CLASSPATH* variable. This class loader is implemented by the *sun.misc.Launcher\$AppClassLoader* class.

The parent-delegation model is a key concept to understand when dealing with class loaders. It states that a class loader delegates class loading to its parent before trying to load the class itself. The parent class loader can be either another custom class loader or the bootstrap class loader. But what is very important is that a class loader can only delegate requests to its parent class loader, never to its child class loaders (it can go up the hierarchy but never down).

The extensions class loader is the parent for the application class loader. The bootstrap class loader is the parent for the extensions class loader. The class loaders hierarchy is shown in Figure 1 on page 3.

If the application class loader needs to load a class, it first delegates to the extensions class loader, which, in turn, delegates to the bootstrap class loader. If the parent class loader cannot load the class, the child class loader tries to find the class in its own repository. In this manner, a class loader is only responsible for loading classes that its ancestors cannot load.

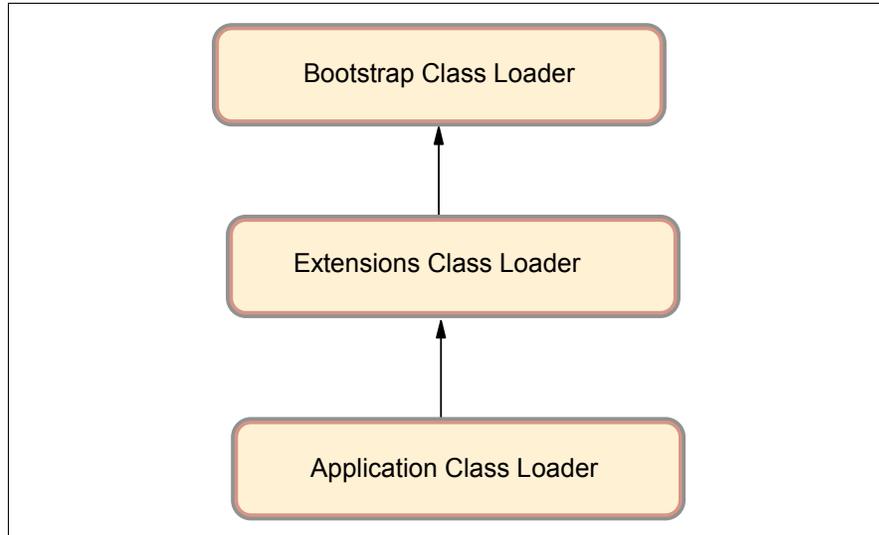


Figure 1 Java class loaders hierarchy

This behavior can lead to some interesting problems if a class is loaded from a class loader that is not on a leaf node in the class loader tree. Consider Example 1. A class called `WhichClassLoader1` loads a class called `WhichClassLoader2`, in turn invoking a class called `WhichClassLoader3`.

Example 1 `WhichClassLoader1` and `WhichClassLoader2` source code

```

public class WhichClassLoader1 {

    public static void main(String[] args) throws
    javax.naming.NamingException {
        // Get classpath values
        String bootClassPath = System.getProperty("sun.boot.class.path");
        String extClassPath = System.getProperty("java.ext.dirs");
        String appClassPath = System.getProperty("java.class.path");

        // Print them out
        System.out.println("Bootstrap classpath =" + bootClassPath +
        "\n");
        System.out.println("Extensions classpath =" + extClassPath +
        "\n");
        System.out.println("Application classpath=" + appClassPath +
        "\n");

        // Load classes
        Object obj = new Object();
  
```

```

WhichClassLoader1 wc11 = new WhichClassLoader1();
WhichClassLoader2 wc12 = new WhichClassLoader2();

// Who loaded what?
System.out.println("Object was loaded by "
    + obj.getClass().getClassLoader());
System.out.println("WCL1 was loaded by "
    + wc11.getClass().getClassLoader());
System.out.println("WCL2 was loaded by "
    + wc12.getClass().getClassLoader());

wc12.getClass();
}
}
=====
===
public class WhichClassLoader2 {

    // This method is invoked from WhichClassLoader1
    public void getClass() {
        WhichClassLoader3 wc13 = new WhichClassLoader3();
        System.out.println("WCL3 was loaded by "
            + wc13.getClass().getClassLoader());
    }
}

```

If all WhichClassLoaderX classes are put on the application class path, the three classes are loaded by the application class loader, and this sample runs just fine. Now suppose that you package the WhichClassLoader2.class file in a JAR file that you store under *Java_home/jre/lib/ext* directory. You can see the output in Example 2.

Example 2 NoClassDefFoundError exception trace

Bootstrap classpath

```

=C:\WebSphere\AppServer\java\jre\lib\vm.jar;C:\WebSphere\AppServer\java\jre\lib\core.jar;C:\WebSphere\AppServer\java\jre\lib\charsets.jar;C:\WebSphere\AppServer\java\jre\lib\graphics.jar;C:\WebSphere\AppServer\java\jre\lib\security.jar;C:\WebSphere\AppServer\java\jre\lib\ibmpkcs.jar;C:\WebSphere\AppServer\java\jre\lib\ibmorb.jar;C:\WebSphere\AppServer\java\jre\lib\ibmcfw.jar;C:\WebSphere\AppServer\java\jre\lib\ibmborbapi.jar;C:\WebSphere\AppServer\java\jre\lib\ibmjcefw.jar;C:\WebSphere\AppServer\java\jre\lib\ibmjgssprovider.jar;C:\WebSphere\AppServer\java\jre\lib\ibmjsseprovider2.jar;C:\WebSphere\AppServer\java\jre\lib\ibmjaaslm.jar;C:\WebSphere\AppServer\java\jre\lib\ibmjaasactive1m.jar;C:\WebSphere\AppServer\java\jre\lib\ibmcertpathprovider.jar;C:\WebSphere\AppServer\ja

```

```
va\jre\lib\server.jar;C:\WebSphere\AppServer\java\jre\lib\xml.jar
Extensions classpath =C:\WebSphere\AppServer\java\jre\lib\ext
Application classpath=.
```

```
Exception in thread "main" java.lang.NoClassDefFoundError:
WhichClassLoader3
    at java.lang.J9VMInternals.verifyImpl(Native Method)
    at java.lang.J9VMInternals.verify(J9VMInternals.java:59)
    at java.lang.J9VMInternals.initialize(J9VMInternals.java:120)
    at WhichClassLoader1.main(WhichClassLoader1.java:17)
```

As you can see, the program fails with a `NoClassDefFoundError` exception, which might sound strange because `WhichClassLoader3` is on the application class path. The problem is that it is *now* on the wrong class path.

What happened was that the `WhichClassLoader2` class was loaded by the extensions class loader. In fact, the application class loader delegated the load of the `WhichClassLoader2` class to the extensions class loader, which in turn delegated the request to the bootstrap class loader. Because the bootstrap class loader could not find the class, the class loading control was returned to the extensions class loader. The extensions class loader found the class on its class path and loaded it.

Now, when a class has been loaded by a class loader, any new classes that the class needs reuse the same class loader to load them (or goes up the hierarchy according to the parent-delegation model). So when the `WhichClassLoader2` class needed to access the `WhichClassLoader3` class, it is the extensions class loader that first gets the request to load it. The extensions class loader first delegates the request to the Bootstrap class path, which cannot find the class, and then tries to load it itself but does not find it either because `WhichClassLoader3` is not on the extensions class path but on the application classpath. And because the extensions class loader cannot delegate the request to the application class loader (a delegate request can only go up the hierarchy, never down), a `NoClassDefFoundError` exception is thrown.

Note: Remember that developers very often also load property files through the class loader mechanism using the following syntax:

```
Properties p = new Properties();
p.load(MyClass.class.getClassLoader().getResourceAsStream("myApp.properties"));
```

This means, if the class `MyClass` is loaded by the extensions class loader and the `myApp.properties` file is only seen by the application class loader, the loading of the property file fails.

WebSphere class loaders overview

Note: Keep in mind when reading the following discussion that each JVM has its own set of class loaders. In a WebSphere environment hosting multiple application servers (JVMs), this means the class loaders for the JVMs are completely separate even if they are running on the same physical machine.

Also note that the JVM uses class loaders called the extensions and application class loaders. As you will see, the WebSphere runtime also uses class loaders called extensions and application class loader, but despite their names, they are not the same as the JVM ones.

WebSphere provides several custom delegated class loaders, as shown in Figure 2.

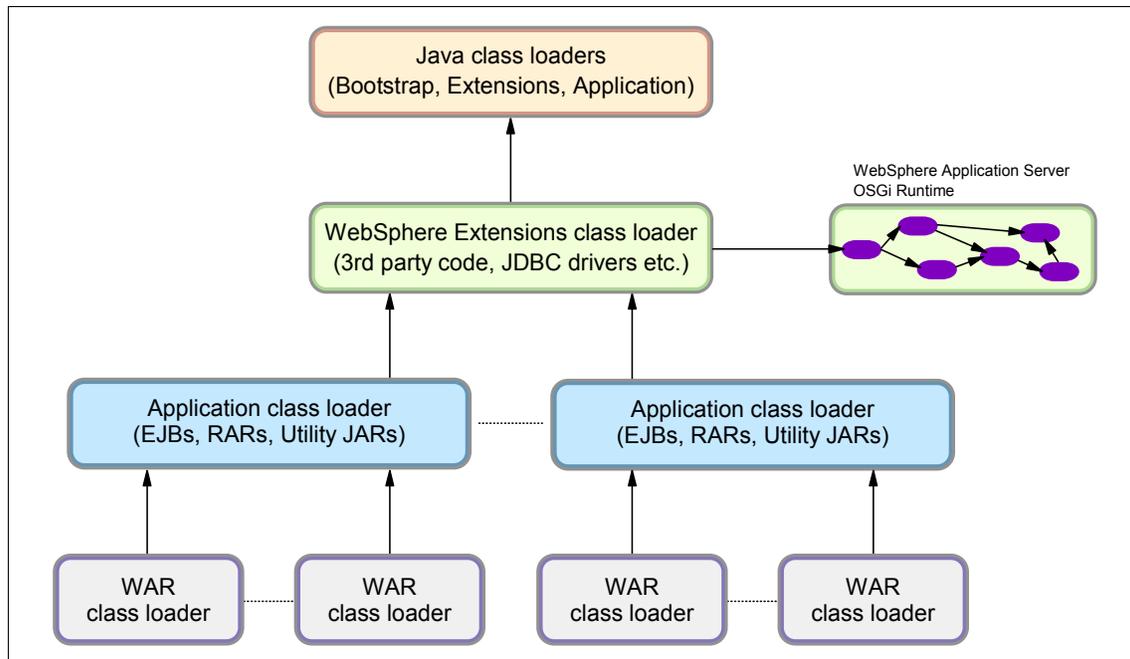


Figure 2 WebSphere class loaders hierarchy

The top box represents the Java (bootstrap, extensions, and application) class loaders. WebSphere loads just enough here to get itself bootstrapped and initialize the WebSphere extensions class loader.

WebSphere extensions class loader

The WebSphere extensions class loader is where WebSphere itself is loaded. In versions of WebSphere prior to V6.1, the runtime was loaded by this single class loader. However, beginning with V6.1, WebSphere is packaged as a set of OSGi bundles. Each OSGi bundle is loaded separately by its own class loader. This network of OSGi class loaders is then connected to the extensions class loader and the rest of the class loader hierarchy through an OSGi gateway class loader.

Despite this architectural change in the internals of how WebSphere loads its own classes, there is no behavioral change as far as your applications are concerned. They still have the same visibility, and the same class loading options still exist for your application.

Prior to V6.1, the WebSphere runtime classes files were stored in the `classes`, `lib`, `lib\ext`, and `installedChannels` directories in the `install_root` directory. Because of the OSGi packaging, these directories no longer exist and the runtime classes are now stored in the `install_root\plugins` directory.

The class path used by the extensions class loader is retrieved from the `ws.ext.dirs` system property, which is initially derived from the `WAS_EXT_DIRS` environment variable set in the `setupCmdLine` script file. The default value of `ws.ext.dirs` is displayed in Example 3.

Example 3 Default value of ws.ext.dirs

```
SET
WAS_EXT_DIRS=%JAVA_HOME%\lib;%WAS_HOME%\classes;%WAS_HOME%\lib;%WAS_HOME%\
E%\installedChannels;%WAS_HOME%\lib\ext;%WAS_HOME%\web\help;%ITP_LOC%\p
lugins\com.ibm.etools.ejbdeploy\runtime
```

Each directory listed in the `ws.ext.dirs` environment variable is added to the WebSphere extensions class loaders class path and every JAR file and ZIP file in the directory is added to the class path.

While the `classes` and `installedChannels` directories no longer exist in the `install_root` directory, the `setupCmdLine` script still adds them to the extensions class path. This means that if you have added your own JAR files to one of these directories in previous releases, you could create this directory and add your JAR files to it and they would still be loaded by the extensions class loader. However, this is not recommended and you should really try to migrate away from such a setup.

On the other hand, if you have developed Java applications that rely on the WebSphere JAR files that were in the `install_root\lib` directory prior to V6.1, you will need to modify your application to retain compatibility. WebSphere

Application Server provides two thin client libraries designed specifically for such applications: one administrative client library and one Web services client library. These thin client libraries can be found in the *install_root*\runtimes directory:

- ▶ com.ibm.ws.admin.client_7.0.0.jar
- ▶ com.ibm.ws.webservices.thinclient_7.0.0.jar

These libraries provide everything your application might need for connecting to and working with WebSphere. WebSphere Application Server V7 gives you the ability to restrict access to internal WebSphere classes so that your applications do not make unsupported calls to WebSphere classes not published in the official WebSphere Application Server API. This setting is an application server setting called **Access to internal server classes**.

The default setting is **Allow**, meaning that your applications can make unrestricted calls to non-public internal WebSphere classes. This function is *not* recommended and might be prohibited in future releases. Therefore, as an administrator, it is a good idea to switch this setting to **Restrict** to see if your applications still work. If they depend on non-public WebSphere internal classes, you will receive a `ClassNotFoundException`, and in that case you can switch back to **Allow**. Your developers should then try to migrate their applications so that they do not make unsupported calls to the WebSphere internal classes in order to retain compatibility with future WebSphere Application Server releases.

Application and Web module class loaders

Java EE 5 applications consist of five primary elements: Web modules, EJB™ modules, application client modules, resource adapters (RAR files), and utility JARs. Utility JARs contain code used by both EJBs and servlets. Utility frameworks such as log4j are good examples of a utility JAR.

EJB modules, utility JARs, resource adapter files, and shared libraries associated with an application are always grouped together into the same class loader. This class loader is called the application class loader. Depending on the class loader policy, this class loader can be shared by multiple applications (EARs), or be unique for each application, which is the default.

By default, Web modules receive their own class loader, a WAR class loader, to load the contents of the WEB-INF/classes and WEB-INF/lib directories. You can modify the default behavior by changing the application's WAR class loader policy. This policy setting can be found in the administrative console by selecting **Applications** → **WebSphere enterprise applications** → *application_name* → **Class loading and update detection** → **WAR class loader policy**. See Figure 3.

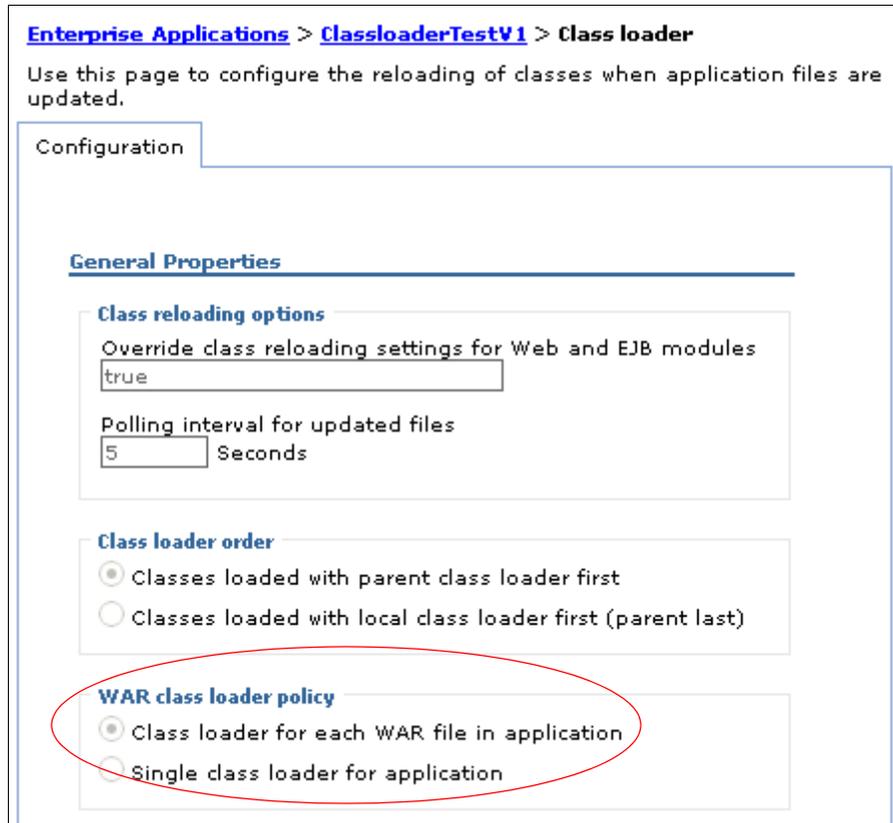


Figure 3 WAR class loader policy

The default is set to **Class loader for each WAR file in the application**. This setting is called **Module** in previous releases and in the application deployment descriptor as viewed in Rational® Application Developer.

If the WAR class loader policy is set to **Single class loader for application**, the Web module contents are loaded by the application class loader in addition to the EJBs, RARs, utility JARs, and shared libraries. The application class loader is the parent of the WAR class loader. This setting is called **Application** in previous releases and in the application deployment descriptor as viewed in Rational Application Developer.

The application and the WAR class loaders are reloadable class loaders. They monitor changes in the application code to automatically reload modified classes. You can modify this behavior at deployment time.

Handling JNI code

Because a JVM only has a single address space, and native code can only be loaded once per address space, the JVM specification states that native code can only be loaded by one class loader in a JVM.

This might cause a problem if, for example, you have an application (EAR file) with two Web modules that both need to load the same native code through a Java Native Interface (JNI™). Only the Web module that first loads the library will succeed.

To solve this problem, you can break out just the few lines of Java code that load the native code into a class on its own and place this file on WebSphere's application class loader (in a utility JAR). However, if you deploy multiple such applications (EAR files) to the same application server, you have to place the class file on the WebSphere extensions class loader instead to ensure that the native code is only loaded once per JVM.

If the native code is placed on a reloadable class loader (such as the application class loader or the WAR class loader), it is important that the native code can properly unload itself should the Java code have to reload. WebSphere has no control over the native code, and if it does not unload and load properly, the application might fail.

If one native library depends on another one, things become even more complicated. For more details, search for Dependent native library in the Information Center.

Configuring WebSphere for class loaders

In the previous topic, you learned about WebSphere class loaders and how they work together to load classes. There are settings in WebSphere Application Server that allow you to influence WebSphere class loader behavior. This section discusses these options.

Application server class loader policies

For each application server in the system, the class loader policy can be set to Single or Multiple. These settings can be found in the administrative console by selecting **Servers** → **Server Types** → **WebSphere application servers** → **server_name**. See Figure 4.

[Application servers](#) > **server1**

Use this page to configure an application server. An application server required to run enterprise applications.

Runtime Configuration

General Properties

Name
server1

Node name
t60Node01

Run in development mode

Parallel start

Start components as needed

Access to internal server classes
Allow

Server-specific Application Settings

ClassLoader policy
Multiple

Class loading mode
Classes loaded with parent class loader first

Figure 4 Application server classloader settings

When the application server class loader policy is set to **Single**, a single application class loader is used to load all EJBs, utility JARs, and shared libraries within the application server (JVM). If the WAR class loader policy then has been set to **Single class loader for application**, the Web module contents for this particular application are also loaded by this single class loader.

When the application server class loader policy is set to **Multiple**, the default, each application will receive its own class loader for loading EJBs, utility JARs, and shared libraries. Depending on whether the WAR class loader policy is set to **Class loader for each WAR file in application** or **Single class loader for application**, the Web module might or might not receive its own class loader.

Here is an example to illustrate. Suppose that you have two applications, Application1 and Application2, running in the same application server. Each application has one EJB module, one utility JAR, and two Web modules. If the application server has its class loader policy set to **Multiple** and the class loader policy for all the Web modules are set to **Class loader for each WAR file in application**, the result is as shown in Figure 5.

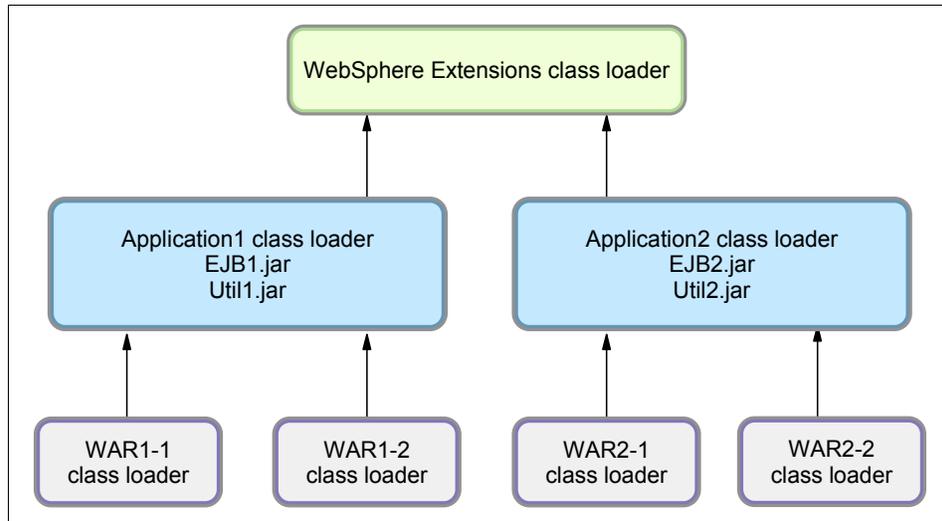


Figure 5 Class loader policies: Example 1

Each application is completely separated from the other and each Web module is also completely separated from the other one in the same application. WebSphere's default class loader policies results in total isolation between the applications and the modules.

If we now change the class loader policy for the WAR2-2 module to **Single class loader for application**, the result is shown in Figure 6.

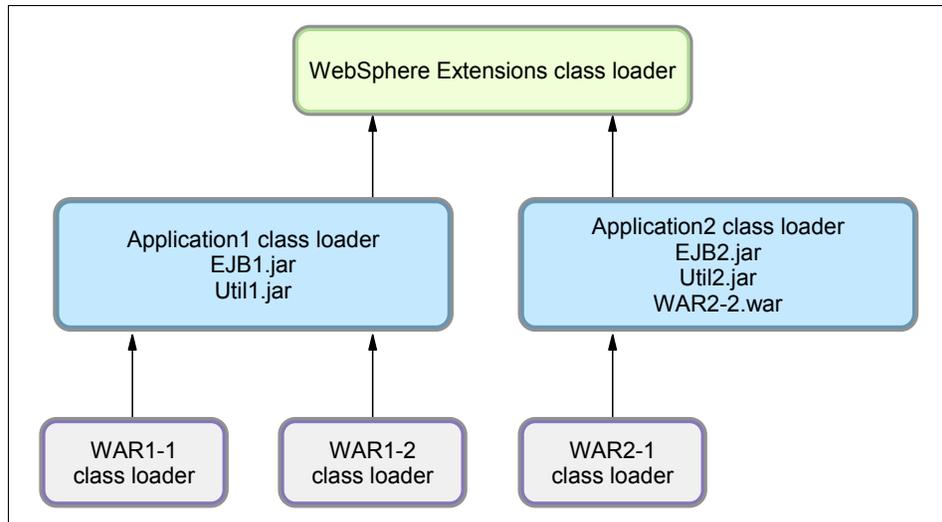


Figure 6 Class loader policies: Example 2

Web module WAR2-2 is loaded by Application2's class loader and classes, and for example, classes in Util2.jar are able to see classes in WAR2-2's /WEB-INF/classes and /WEB-INF/lib directories.

As a last example, if we change the class loader policy for the application server to **Single** and also change the class loader policy for WAR2-1 to **Single class loader for application**, the result is as shown in Figure 7.

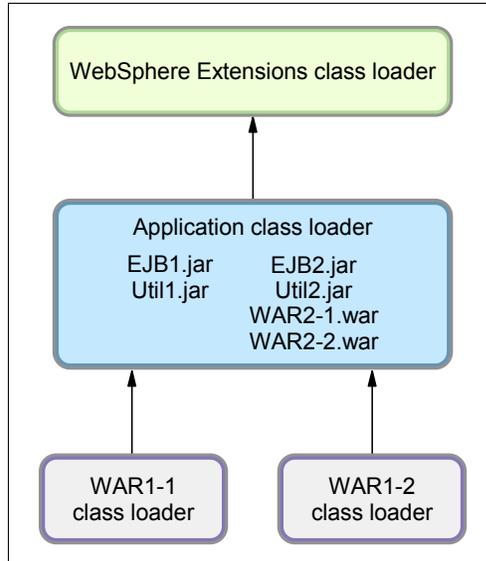


Figure 7 Class loader policies: Example 3

There is now only a single application class loader loading classes for both Application1 and Application2. Classes in Util1.jar can see classes in EJB2.jar, Util2.jar, WAR2-1.war and WAR2-2.war. The classes loaded by the application class loader still cannot, however, see the classes in the WAR1-1 and WAR1-2 modules, because a class loader can only find classes by going up the hierarchy, never down.

Class loading/delegation mode

WebSphere's application class loader and WAR class loader both have a setting called the class loader order (see Figure 4 on page 11). This setting determines whether the class loader order should follow the normal Java class loader delegation mechanism, as described in "A brief introduction to Java class loaders" on page 2, or override it.

There are two possible options for the class loading mode:

- ▶ Classes loaded with parent class loader first
- ▶ Classes loaded with local class loader first (parent last)

In previous WebSphere releases, these settings were called PARENT_FIRST and PARENT_LAST, respectively.

The default value for class loading mode is **Classes loaded with parent class loader first**. This mode causes the class loader to first delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. This is the default policy for standard Java class loaders.

If the class loading policy is set to **Classes loaded with local class loader first (parent last)**, the class loader attempts to load classes from its local class path before delegating the class loading to its parent. This policy allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

Note: The administrative console is a bit confusing at this point. On the settings page for a Web module, the two options for class loader order are Classes loaded with parent class loader first and Classes loaded with local class loader first (parent last). However, in this context, the “local class loader” really refers to the WAR class loader, so the option **Classes loaded with local class loader first** should really be called Classes loaded with WAR class loader first.

Assume that you have an application, similar to Application1 in the previous examples, and it uses the popular log4j package to perform logging from both the EJB module and the two Web modules. Also assume that each module has its own, unique, log4j.properties file packaged into the module. You could configure log4j as a utility JAR so you would only have a single copy of it in your EAR file.

However, if you do that, you might be surprised to see that all modules, including the Web modules, load the log4j.properties file from the EJB module. The reason is that when a Web module initializes the log4j package, the log4j classes are loaded by the application class loader. Log4j is configured as a utility JAR. Log4j then looks for a log4j.properties file on its class path and finds it in the EJB module.

Even if you do not use log4j for logging from the EJB module and the EJB module does not, therefore, contain a log4j.properties file, log4j does not find the log4j.properties file in any of the Web modules anyway. The reason is that a class loader can only find classes by going up the hierarchy, never down.

To solve this problem, you can use one of the following approaches:

- ▶ Create a separate file, for example, Resource.jar, configure it as a utility JAR, move all log4j.properties files from the modules into this file, and make their names unique (like war1-1_log4j.properties, war1-2_log4j.properties, and ejb1_log4j.properties). When initializing log4j from each module, tell it to load the proper configuration file for the module instead of the default (log4j.properties).

- ▶ Keep the log4j.properties for the Web modules in their original place (/WEB-INF/classes), add log4j.jar to both Web modules (/WEB-INF/lib) and set the class loading mode for the Web modules to **Classes loaded with local class loader first (parent last)**. When initializing log4j from a Web module, it loads the log4j.jar from the module itself and log4j would find the log4j.properties on its local classpath, the Web module itself. When the EJB module initializes log4j, it loads from the application class loader and it finds the log4j.properties file on the same class path, the one in the EJB1.jar file.
- ▶ If possible, merge all log4j.properties files into one and place it on the Application class loader, in a Resource.jar file, for example).

Singletons: The Singleton pattern is used to ensure that a class is instantiated only once. However, *once* only means *once for each class loader*. If you have a Singleton being instantiated in two separated Web modules, two separate instances of this class will be created, one for each WAR class loader. So in a multi-class loader environment, special care must be taken when implementing Singletons.

Shared libraries

Shared libraries are files used by multiple applications. Examples of shared libraries are commonly used frameworks like Apache Struts or log4j. You use shared libraries typically to point to a set of JARs and associate those JARs to an application, a Web module, or the class loader of an application server. Shared libraries are especially useful when you have different versions of the same framework you want to associate to different applications.

Shared libraries are defined using the administration tools. They consist of a symbolic name, a Java class path, and a native path for loading JNI libraries. They can be defined at the cell, node, server, or cluster level. However, simply defining a library does not cause the library to be loaded. You must associate the library to an application, a Web module, or the class loader of an application server for the classes represented by the shared library to be loaded. Associating the library to the class loader of an application server makes the library available to all applications on the server.

Note: If you associate a shared library to an application, do not associate the same library to the class loader of an application server.

You can associate the shared library to an application in one of two ways:

- ▶ You can use the administrative console. The library is added using the **Shared libraries references** link under the References section for the enterprise application.
- ▶ You can use the manifest file of the application and the shared library. The shared library contains a manifest file that identifies it as an extension. The dependency to the library is declared in the application's manifest file by listing the library extension name in an extension list.

For more information about this method, search for installed optional packages in the Information Center.

Shared files are associated with the class loader of an application server using the administrative tools. The settings are found in the Server Infrastructure section. Expand the Java and Process Management. Select **Class loader** and then click the **New** button to define a new class loader. After you have defined a new class loader, you can modify it and, using the Shared library references link, you can associate it to the shared libraries you need.

See “Step 4: Sharing utility JARs using shared libraries” on page 26 for more details.

Class loader viewer

If the Class Loader Viewer Service is not enabled, the Class Loader Viewer only displays the hierarchy of class loaders and their classpaths, but not the classes actually loaded by each of the class loaders. This also means that the search capability of the Class Loader Viewer is lost.

To enable the Class Loader Viewer Service, select **Servers** → **Server Types** → **WebSphere application server** → *server_name* and then click the **Class Loader Viewer Service** under the **Additional Properties** link. Then select **Enable service at server startup**. You will need to restart the application server for the setting to take effect.

In the next section, we give you an example of how to work with the different class loader settings, then we also use the Class Loader Viewer to illustrate the different results.

Learning class loaders by example

We have now described all the different options for influencing class loader behavior. In this section, we take an example and use all the different options we have discussed to this point so that you can better evaluate the best solution for your applications.

We have created a very simple application, with one servlet and one EJB. Both call a class, `VersionChecker`, shown in Example 4. This class can print which class loader was used to load the class. The `VersionChecker` class also has an internal value that can be printed to check which version of the class we are using. This will be used later to demonstrate the use of multiple versions of the same utility JAR.

Example 4 VersionChecker class source code

```
package com.itso.classloaders;

public class VersionChecker {
    static final public String classVersion = "v1.0";

    public String getInfo() {
        return ("VersionChecker is " + classVersion +
            ". Loaded by " + this.getClass().getClassLoader());
    }
}
```

After being installed, the application can be invoked through `http://localhost:9080/ClassLoaderExampleWeb/ExampleServlet`. This invokes the `ExampleServlet` which calls `VersionChecker` and then displays the classloader.

The `VersionCheckerV1.jar` file contains the `VersionChecker` class file that returns Version number 1.0. For all the following tests, we have, unless otherwise noted, left the class loader policies and loading modes to their defaults. In other words, we have one class loader for the application and one for the WAR file. Both have their delegation modes set to **Classes loaded with parent class loader first**.

Step 1: Simple Web module packaging

Start with the following assumption: our utility class is only used by a servlet. We have placed the VersionCheckerV1.jar file under the WEB-INF/lib directory of the Web module.

Tip: You place JAR files used by a single Web module, or a JAR file that *only* this Web module should see under WEB-INF/lib.

When we run the application in such a configuration, we obtain the results shown in Example 5.

Example 5 Class loader: Example 1

```
VersionChecker called from Servlet
VersionChecker is v1.0.
Loaded by
com.ibm.ws.classloader.CompoundClassLoader@18721872[war:ClassLoaderExample/ClassLoaderExampleWeb.war]
```

Local ClassPath:

```
C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExample.ear\ClassLoaderExampleWeb.war\WEB-INF\classes;
C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExample.ear\ClassLoaderExampleWeb.war\WEB-INF\lib\VersionCheckerV1.jar;
C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExample.ear\ClassLoaderExampleWeb.war
```

```
Parent:
com.ibm.ws.classloader.CompoundClassLoader@7f277f27[app:ClassLoaderExample]
Delegation Mode: PARENT_FIRST
```

There are a few things that we can learn from this trace:

1. The type of the WAR class loader is:
com.ibm.ws.classloader.CompoundClassLoader
2. It searches classes in the following order:
ClassLoaderExampleWeb.war\WEB-INF\classes
ClassLoaderExampleWeb.war\WEB-INF\lib\VersionCheckerV1.jar
ClassLoaderExampleWeb.war

The WEB-INF/classes folder holds unpacked resources (such as servlet classes, plain Java classes, and property files), while the WEB-INF/lib holds resources packaged as JAR files. You can choose to package your Java code in JAR files and place them in the lib directory or you can put them unpacked in the classes directory. They will both be on the same classpath. Because our sample application was developed and exported from the Rational Application Developer, our servlet goes into the classes folder, because the Java classes are not packaged in a JAR file when exporting an application.

The root of the WAR file is the next place where you can put code or properties, but you really should not do that because that folder is the document root for the Web server (if the File Serving Servlet capabilities are enabled, which they are by default) so anything that is in that folder is accessible from a browser. According to the Java EE 5 specification, though, the WEB-INF folder is protected, which is why the classes and lib folders are under WEB-INF.

The class loader class path is dynamically built at application startup.

We can now also use the Class Loader Viewer to display the class loader. In the administrative console, select **Troubleshooting** → **Class Loader Viewer**. Then expand **server1** → **Applications** → **ClassloaderExample** → **Web modules** and click the **ClassloaderExampleWeb.war**, as shown in Figure 8.

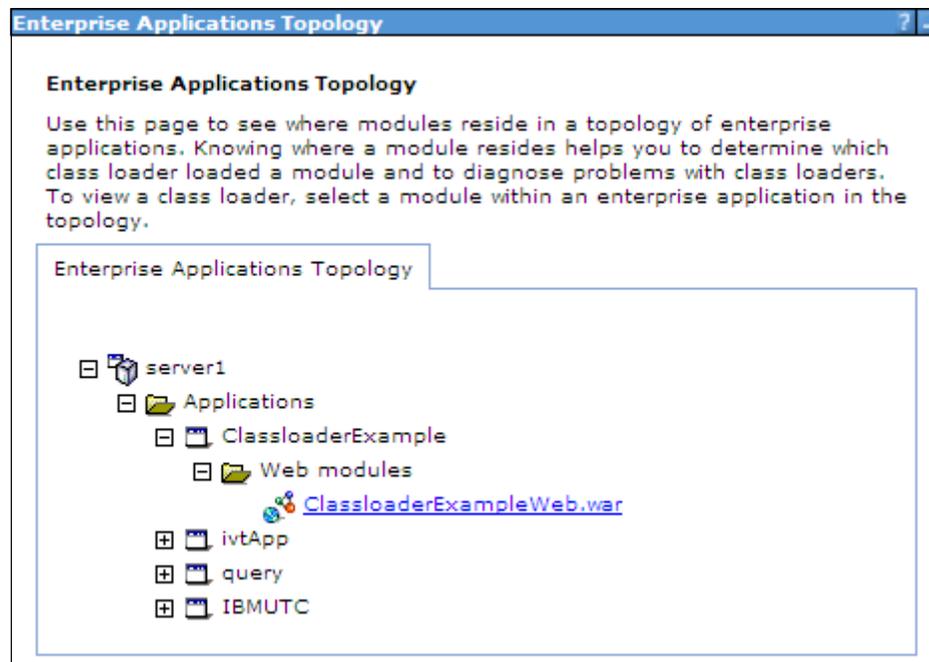


Figure 8 Class Loader Viewer showing applications tree

When the Web module is expanded, the Class Loader Viewer shows the hierarchy of class loaders all the way from the JDK™ Extensions and JDK application class loaders at the top to the WAR class loader at the bottom, called the compound class loader. See Figure 9.

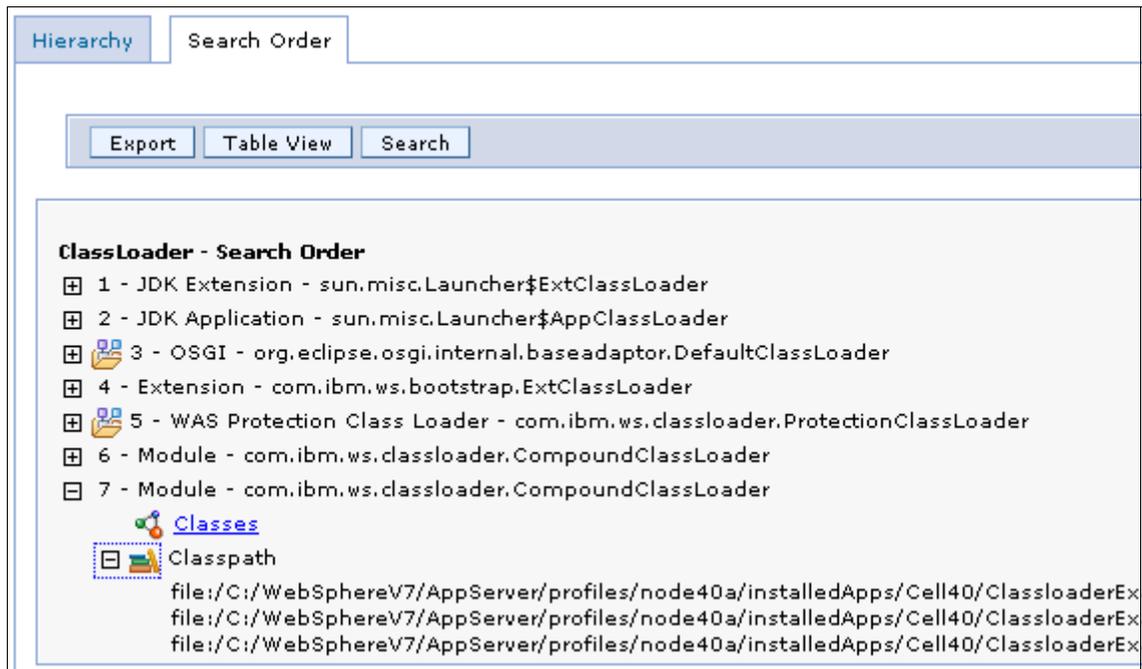


Figure 9 Class Loader Viewer showing class loader hierarchy

If you expand the classpath for `com.ibm.ws.classloader.CompoundClassLoader`, you see the same information as the `VersionChecker` class prints (see Example 5 on page 12).

Note: For the Class Loader Viewer to display the classes loaded, the Class Loader Viewer Service must be enabled as described in “Class loader viewer” on page 17.

The Class Loader Viewer also has a table view that displays all the class loaders and the classes loaded by each of them on a single page. The table view also displays the Delegation mode. True means that classes are loaded with parent class loader first, while false means that classes are loaded with local class loader first (parent last), or the WAR class loader in the case of a Web module. See Figure 10.

Module - com.ibm.ws.classloader.CompoundClassLoader	
Delegation	true
Classpath	file:/C:/WebSphereV7/AppServer/profiles/node40a/installedApps/Cell40/ClassLoaderExample.ear/ClassLoaderExampleWeb.war/WEB-INF/classes
	file:/C:/WebSphereV7/AppServer/profiles/node40a/installedApps/Cell40/ClassLoaderExample.ear/ClassLoaderExampleWeb.war/WEB-INF/lib/VersionCheckerV1.jar
	file:/C:/WebSphereV7/AppServer/profiles/node40a/installedApps/Cell40/ClassLoaderExample.ear/ClassLoaderExampleWeb.war

Figure 10 Class Loader Viewer table view

As you can see, the WAR class loader has loaded our example servlet and the VersionChecker class, just as expected.

The Class Loader Viewer also has a search feature where you can search for classes, JAR files, folders, and so on. This can be particularly useful if you do not know which of the class loaders loaded a class you are interested in. The search feature is case sensitive but allows wild cards, so a search for **VersionChecker** finds our VersionChecker class.

Step 2: Adding an EJB module and utility jar

Next, we decided to add an EJB to our application that also depends on our VersionChecker JAR file. For this task, we added a VersionCheckerV2.jar file to the root of our EAR. The VersionChecker class in this JAR file returns Version 2.0. To make it available as a utility JAR on the extensions class loader, we added a reference to it in the EJB module's manifest file, as shown in Example 6.

Example 6 Updated MANIFEST.MF for EJB module

```
Manifest-Version: 1.0
Class-Path: VersionCheckerV2.jar
```

The result is that we now have a Web module with a servlet in the WEB-INF/classes folder and the VersionCheckerV1.jar file in the WEB-INF/lib folder. We also have an EJB module that references the VersionCheckerV2.jar utility JAR in the root of the EAR. Which version of the VersionChecker class file would you expect the Web module to load? Version 1.0 from the WEB-INF/lib or version 2.0 from the utility JAR?

The test results are shown in Example 7.

Example 7 Class loader: Example 2

VersionChecker called from Servlet

VersionChecker is v2.0.

Loaded by

com.ibm.ws.classloader.CompoundClassLoader@404a404a[app:ClassLoaderExampleV2]

Local ClassPath:

C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\ClassLoaderExampleEJB.jar;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\VersionCheckerV2.jar

Parent: com.ibm.ws.classloader.ProtectionClassLoader@a540a54

Delegation Mode: PARENT_FIRST

VersionChecker called from EJB

VersionChecker is v2.0.

Loaded by

com.ibm.ws.classloader.CompoundClassLoader@404a404a[app:ClassLoaderExampleV2]

Local ClassPath:

C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\ClassLoaderExampleEJB.jar;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\VersionCheckerV2.jar

Parent: com.ibm.ws.classloader.ProtectionClassLoader@a540a54

Delegation Mode: PARENT_FIRST

As you can see, the VersionChecker is Version 2.0 when called both from the EJB module and the Web module. The reason is, of course, that the WAR class loader delegates the request to its parent class loader instead of loading it itself, so the Utility JAR is loaded by the same class loader regardless of whether it was called from the servlet or the EJB.

Step 3: Changing the WAR class loader delegation mode

What if we now wanted the Web module to use the VersionCheckerV1.jar file from the WEB-INF/lib folder? For that task, we would have to change the class loader delegation from parent first to parent last.

Set the delegation mode to PARENT_LAST, using the following steps:

1. Select the **WebSphere Enterprise Applications** entry in the navigation area.
2. Select the **ClassLoaderExample** application.
3. Select **Manage modules** under the Modules section.
4. Select the **ClassLoaderExampleWeb** module.
5. Change the Class loader order to **Classes loaded with local class loader first (parent_last)**. Remember, this entry should really be called Classes loaded with WAR class loader first, as noted in “Class loading/delegation mode” on page 14.
6. Click **OK**.
7. Save the configuration.
8. Restart the application.

The VersionCheckerV1 in WEB-INF/lib returns a class version of 1.0. You can see in Example 8 that this is the version now used by the WAR file.

Example 8 Class loader: Example 3

**VersionChecker called from Servlet
VersionChecker is v1.0.**

Loaded by

```
com.ibm.ws.classloader.CompoundClassLoader@1c421c42[war:ClassLoaderExampleV2/ClassLoaderExampleWeb.war]
```

Local ClassPath:

```
C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\ClassLoaderExampleWeb.war\WEB-INF\classes;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\ClassLoaderExampleWeb.war\WEB-INF\lib\VersionCheckerV1.jar;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\ClassLoaderExampleWeb.war
```

Parent:

```
com.ibm.ws.classloader.CompoundClassLoader@1a5e1a5e[app:ClassLoaderExampleV2]
```

Delegation Mode: PARENT_LAST

VersionChecker called from EJB

VersionChecker is v2.0.

Loaded by

com.ibm.ws.classloader.CompoundClassLoader@1a5e1a5e[app:ClassLoaderExampleV2]

Local ClassPath:

C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\ClassLoaderExampleEJB.jar;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Ce1140\ClassLoaderExampleV2.ear\VersionCheckerV2.jar

Parent: com.ibm.ws.classloader.ProtectionClassLoader@a540a54

Delegation Mode: PARENT_FIRST

Tip: Use this technique to specify that a Web module should use a specific version of a library, such as Struts, or to override classes coming with the WebSphere runtime. Put the common version at the top of the hierarchy, and the specialized version in WEB-INF/lib.

The Java EE 5 specification does not provide a standard option to specify the delegation mode in the EAR file. However, by using a WebSphere Extended EAR file, you can specify this setting so you do not have to change it every time you redeploy your application.

If you use the search feature of the Class Loader Viewer to search for “*VersionChecker*”, you would see the two entries in Example 9.

Example 9 Class Loader Viewer search feature

WAS Module Compound Class Loader (WAR class loader):

file: / C: / WebSphereV7 / AppServer / profiles / node40a / installedApps / Ce1140 / ClassloaderExampleV2.ear / ClassloaderExampleWeb.war / WEB-INF / lib / VersionCheckerV1.jar

WAS Module Jar Class Loader (Application class loader):

file: / C: / WebSphereV7 / AppServer / profiles / node40a / installedApps / Ce1140 / ClassloaderExampleV2.ear / VersionCheckerV2.jar

Step 4: Sharing utility JARs using shared libraries

In this situation, the VersionCheckerV2.jar file is used by a single application. What if you wanted to share it among multiple applications? Of course, you could package it within each EAR file. But changes to this utility JAR file require redeploying all applications again. To avoid this, you can externalize global utility JARs using a shared library.

Shared libraries can be defined at the cell, node, application server, and cluster levels. After you have defined a shared library, you must associate it to the class loader of an application server, an application, or an individual Web module. Depending on the target the shared library is assigned to, WebSphere will use the appropriate class loader to load the shared library.

You can define as many shared libraries as you want. You can also associate multiple shared libraries with an application, Web module, or application server.

Using shared libraries at the application level

To define a shared library named VersionCheckerV2_SharedLib and associate it to our ClassloaderTest application, do the following steps:

1. In the administrative console, select **Environment** → **Shared Libraries**.
2. Select the scope at which you want this shared library to be defined, such as Cell, and click **New**.
3. Specify the properties shown in Figure 11.

General Properties

* Scope
cells:Cell40

* Name
VersionCheckerV2_SharedLib

Description

* Classpath
C:\Documents and Settings\Administrator\My Documents\0-Working files\Admin and Config\Downloads\VersionCheckerV2.jar

Native Library Path

Specifies a native library.

Class Loading
 Use an isolated class loader for this shared library

Apply OK Reset Cancel

Figure 11 Shared library configuration

- Name: Enter VersionCheckerV2_SharedLib.
- Class path: Enter the list of entries on the class path. Press Enter between each entry. Note that if you need to provide an absolute path, we highly recommend that you use WebSphere variables, such as %FRAMEWORK_JARS%/VersionCheckerV2.jar. Make sure that you declare this variable at the same scope as the shared library for cell, node, server, or cluster.

- Native library path: Enter a list of DLLs and .so files for use by the JNI code.
 - **(NEW in V7)** If you want to have only one instance of a version of a class shared among applications, select **Use an isolated class loader for this shared library**.
4. Click **OK**.
 5. Select **Applications** → **Application Types** → **WebSphere enterprise applications**.
 6. Select the **ClassloadersExample** application.
 7. In References, select **Shared library references**.
 8. Select **ClassLoaderExample** in the Application row.
 9. Click **Reference shared libraries**.
 10. Select the **VersionCheckerV2_SharedLib** and click the >> button to move it to the Selected column, as shown in Figure 12.

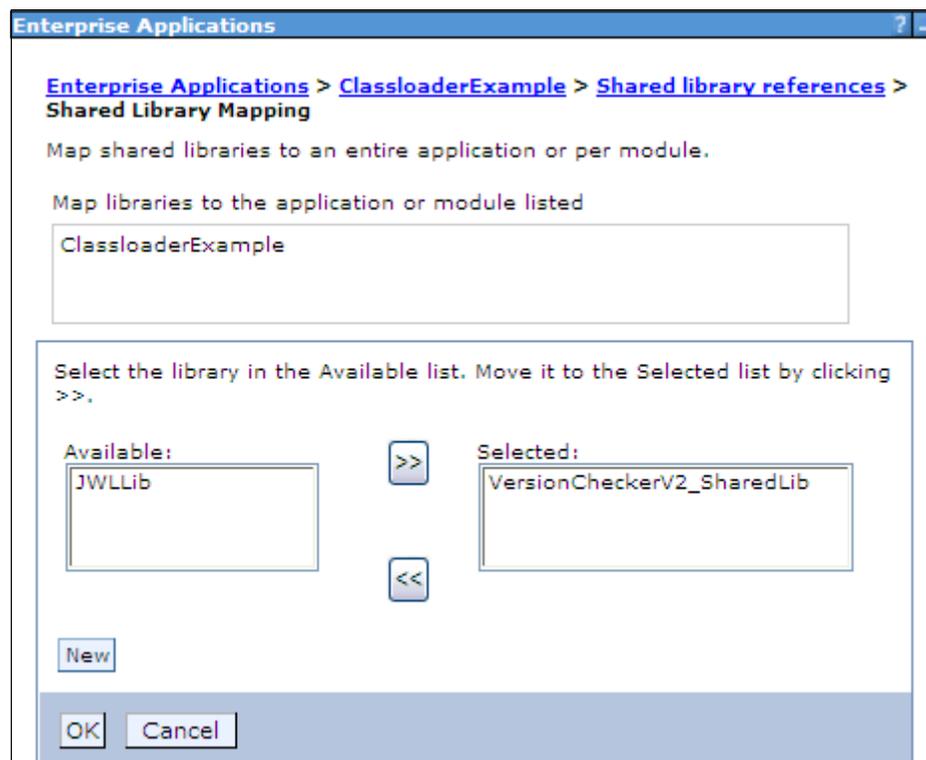


Figure 12 Assigning a shared library

11. Click **OK**.

12. The shared library configuration window for the ClassloaderExample application should now look like Figure 13.

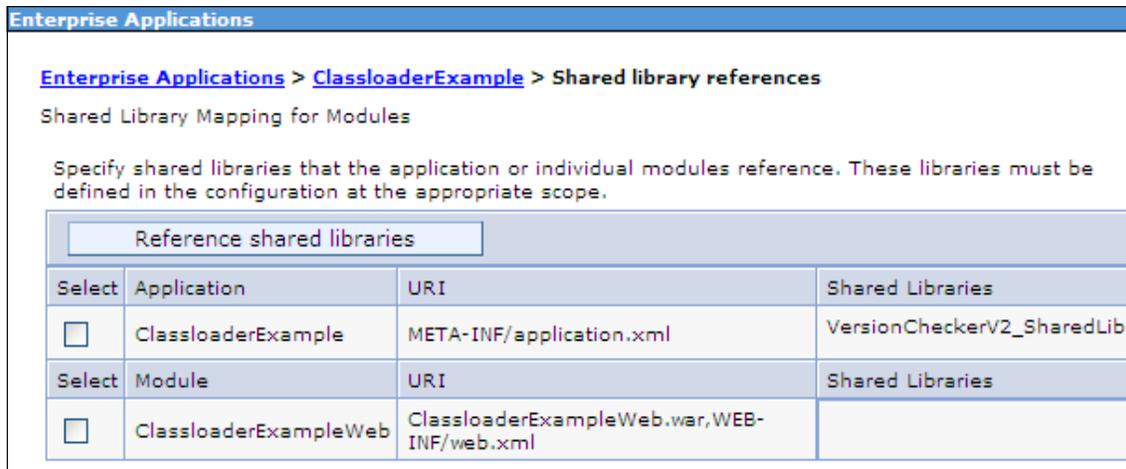


Figure 13 Shared library assigned to ClassloaderExample application

13. Click **OK** and save the configuration.

If we now remove the VersionCheckerV2.jar file from the root of the EAR file, and remove the reference to it from the EJB module's manifest file, and restart the application server, we see the results in Example 10. Remember the class loader order for the Web module is still **Classes loaded with local class loader first (parent last)**.

Example 10 Class loader: Example 5

**VersionChecker called from Servlet
VersionChecker is v1.0.**

Loaded by

com.ibm.ws.classloader.**CompoundClassLoader@405d405d**[war:ClassloaderExampleV3/ClassloaderExampleWeb.war]

Local ClassPath:

C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassloaderExampleV3.ear\ClassloaderExampleWeb.war\WEB-INF\classes;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassloaderExampleV3.ear\ClassloaderExampleWeb.war\WEB-INF\lib\VersionCheckerV1.jar;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassloaderExampleV3.ear\ClassloaderExampleWeb.war Parent:

com.ibm.ws.classloader.**CompoundClassLoader@3e793e79**[app:ClassloaderExampleV3]

Delegation Mode: PARENT_LAST

VersionChecker called from EJB

VersionChecker is v2.0.

Loaded by

com.ibm.ws.classloader.CompoundClassLoader@3e793e79[app:ClassLoaderExampleV3]

Local ClassPath:

C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExampleV3.ear\ClassLoaderExampleEJB.jar;C:\Documents and Settings\Administrator\My Documents\0-Working files\Admin and Config\Downloads\VersionCheckerV2.jar Parent:
com.ibm.ws.classloader.ProtectionClassLoader@7d677d67

Delegation Mode: PARENT_FIRST

As expected, because of the delegation mode for the Web module, the VersionCheckerV1.jar file was loaded when the servlet needed the VersionChecker class. When the EJB needed the VersionChecker class, it was loaded from the shared library, which points to C:\henrik\VersionCheckerV2.jar.

If we would like the Web module to also use the shared library, we would just restore the class loader order to the default, **Classes loaded with parent class loader first**, for the Web module.

Using shared libraries at the application server level

A shared library can also be associated with an application server. All applications deployed on this server see the code listed on that shared library. To associate a shared library to an application server, you must first create an additional class loader for the application server, as follows:

1. Select an application server.
2. In the Server Infrastructure section, expand the **Java and Process Management**. Select **Class loader**.
3. Choose **New**, and select a class loader order for this class loader, Classes loaded with parent class loader first or Classes loaded with local class loader first (parent last). Click **OK**.
4. Click the class loader that is created.
5. Click **Shared library references**.
6. Click **Add**, and select the library you want to associate to this application server. Repeat this operation to associate multiple libraries to this class

loader. For our example, we selected the **VersionCheckerV2_SharedLib** entry.

7. Click **OK**.
8. Save the configuration.
9. Restart the application server for the changes to take effect.

Because we have now attached the VersionCheckerV2 shared library to the class loader of the application server, we obtain the results in Example 11.

Example 11 Class loader: Example 6

VersionChecker called from Servlet
VersionChecker is v1.0.

Loaded by
com.ibm.ws.classloader.**CompoundClassLoader@26a426a4**[war:ClassLoaderExampleV3/ClassLoaderExampleWeb.war]

Local ClassPath:
C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExampleV3.ear\ClassLoaderExampleWeb.war\WEB-INF\classes;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExampleV3.ear\ClassLoaderExampleWeb.war\WEB-INF\lib\VersionCheckerV1.jar;C:\WebSphereV7\AppServer\profiles\node40a\installedApps\Cell140\ClassLoaderExampleV3.ear\ClassLoaderExampleWeb.war Parent:
com.ibm.ws.classloader.CompoundClassLoader@1f381f38[app:ClassLoaderExampleV3]

Delegation Mode: **PARENT_LAST**

VersionChecker called from EJB
VersionChecker is v2.0.

Loaded by com.ibm.ws.classloader.**ExtJarClassLoader@48964896**[server:0]

Local ClassPath: **C:\Documents and Settings\Administrator\My Documents\0-Working files\Admin and Config\Downloads\VersionCheckerV2.jar**
Parent: com.ibm.ws.classloader.ProtectionClassLoader@7d677d67
Delegation Mode: **PARENT_FIRST**

The new class loader we defined is called the ExtJarClassLoader and it has loaded the VersionCheckerV2.jar file when requested by the EJB module.

The WAR class loader still continues to load its own version due to the delegation mode.

Tip: For further details, tuning, tuning and troubleshooting, see the IBM® JDK6 Diagnostics Guide, found in DeveloperWorks at:

<http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html>

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

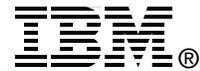
COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright International Business Machines Corporation 2009. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This document REDP-4581-00 was created or updated on October 12, 2009.



Send us your comments in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:
ibm.com/redbooks
- ▶ Send your comments in an email to:
redbook@us.ibm.com
- ▶ Mail your comments to:
IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099, 2455 South Road
Poughkeepsie, NY 12601-5400 U.S.A.



Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

IBM®
Rational®

Redbooks (logo) ®
WebSphere®

The following terms are trademarks of other companies:

EJB, Java, JDK, JNI, JVM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.