Class Loading Conflicts in JVM

This guide shows how to analyze and avoid potential problems caused by class loading conflicts. The content is structured in the following sections, the first one gives a little introduction to the Classloader model in Weblogic [1], following the installation of Classloader Analysis Tool (CAT) [1] is shown and then two examples that shows the use of CAT on a real application is presented.

Class loading in Weblogic

Summarizing the class loading process in just some lines is hard so in this guide the focus is talking about hierarchies and the delegation model.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchies</td>
<td>The classloader in an application server such as Weblogic is based on the model defined by the JVM, which means a hierarchical model that on Weblogic is organized as a tree with these levels: bootstrap class loader, extension class loader, system classpath classloader, application specific classloaders (this includes Weblogic server) [1]. In the previous tree, bootstrap class loader is the root and application specific classloaders are the leaves [1].</td>
</tr>
<tr>
<td>Delegation model</td>
<td>A common question that arises when Java application servers are used is why is my application using the wrong class? This is because the delegation model, which states “The classloader implementation first checks its cache to see if the requested class has already been loaded. This class verification improves performance in that its cached memory copy is used instead of repeated loading of a class from disk. If the class is not found in its cache, the current classloader asks its parent for the class. Only if the parent cannot load the class does the classloader attempt to load the class. If a class exists in both the parent and child classloaders, the parent version is loaded” [1]</td>
</tr>
</tbody>
</table>

The following picture depicts this hierarchy of classloaders.
Java Classloader Hierarchy

Java Virtual Machine

Create

Bootstrap class loader

[Root]

Loads in memory Java Development Kit (JDK) **internal** classes and java.* packages that belong to the JVM.

Extension class loader

Extends

Loads in memory any JAR placed in the JDK’s extension directory.

System classpath classloader

Extends

Loads in memory the classes from the classpath of the JVM.

Application specific classloaders (this includes Weblogic server)
[Leaves]

Loads in memory the classes brought by the application (EAR, War file)

The hierarchy of classloaders in Java

In the previous figure let us say one application specific classloader wants to load a class so the following diagram depicts this process.
In the previous diagram is possible to see that because of the delegation process even if the application specific classloader has the requested class, this will be loaded by a superior classloaders in the hierarchy if the class exists on one of the superior levels otherwise the class is loaded by the application specific classloader.
How could we subvert the previous process?

Of course, there are ways to subvert this process to allow us using our own libraries as is described in the following table.

<table>
<thead>
<tr>
<th>Way to subvert the process</th>
<th>Definition</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
</table>
| prefer-web-inf-classes Element | “If true, classes located in the WEB-INF directory of a web-app will be loaded in preference to classes loaded in the application or system classloader” [1] | - This is the easiest way to subvert the delegation model to use your own classes. <xml version="1.0" encoding="UTF-8">  
  <!DOCTYPE weblogic-web-app SYSTEM "http://www.bea.com/servers/weblogic-web-app.dtd">  
  <weblogic-web-app>  
    <container-descriptor>  
      <prefer-web-inf-classes>true</prefer-web-inf-classes>  
    </container-descriptor>  
  </weblogic-web-app> | - With this you can include some undesirable classes that are part of the library because a library can include many packages and classes so you should know the library thoroughly before subverting the delegation model in this way.  
- This is prone to be affected by new bugs introduced on latest version of JVMs as can be seen in these two bugs [3] and [4].  
- Sometimes can be unpredictable as can be seen in the second example shown below on this document. |
| Using a Filtering ClassLoader | This is mechanism to use third party libraries telling the class loader which packages are going to be loaded by the application classloader rather than the system classloader [1]. | - The implementation could be a little difficult because each needed package should be mentioned specifically.  
- The risk of including specific undesirable classes is minimized because it specifies packages instead of the whole jar.  
- It looks more stable to face possible bugs since even is recommended as a work around for the bug described in [4] | - The implementation can be tricky because we have to provide details about each package we want our application to use. |
Installing Classloader Analysis Tool (CAT)

This are the steps to install CAT, which is a file called wls-cat.war located on $WL_HOME/server/lib/wls-cat.war

1. After login into the console with an admin user, lock the console to edit and click on Install
2. Look for the application $WL_HOME/server/lib/wls-cat.war and click on Next.
3. Choose the remarked radio button and click on Next

4. Choose the cluster and click on Next
5. Choose the remarked option and click on Next

6. Select the remarked option and click on Finish
7. Click on Activate Changes

8. Go to Deployments > Control and start the application
9. Test the application using the listen address and the port assigned to each managed server inside the cluster. The application will request a user and you should use the **Weblogic** user, after login this is the main page of CAT.
Using CAT to analyze class conflict

In this section two examples related to class conflicts are shown.

**Example 1**

The application CAT identifies class conflicts.
As an example we can analyse oracle.jdbc.* to be specific the class called `oracle.jdbc.connector.OracleLocalTransaction` as can be seen in the following report generated by CAT.

**Resource: oracle.jdbc.connector.OracleLocalTransaction**

**Checksum:** 1f8d1e637d6813c0d486ff626c60f1d2  
**Load Location:** jar:file: $WL_HOME/oracle_common/modules/oracle.jdbc/ojdbc7.jar!/oracle/jdbc/connector/OracleLocalTransaction.class  
**Classloader Type:** com.oracle.classloader.weblogic.LaunchClassLoader  
**Classloader Hash Code:** 572145572  
**Classloader Search Order:** 318781939 -> 572145572  
**Alternative Locations:**

$DOMAIN_HOME/servers/ServerExample/tmp/_WL_user/yyy.xxx_war/2eqtxp/war/WEB-INF/lib/ojdbc7-12.1.0.2.0.jar!/oracle/jdbc/connector/OracleLocalTransaction.class

In the previous figure two class loaders are identified: 318781939 -> 572145572, which means both have the class, but only one of them loads the class. According to the previous report, this class can be loaded from two locations.

**Location:**

jar:file: $WL_HOME/oracle_common/modules/oracle.jdbc/ojdbc7.jar!/oracle/jdbc/connector/OracleLocalTransaction.class

**Alternative Locations:**

$DOMAIN_HOME/servers/ServerExample/tmp/_WL_user/yyy.xxx_war/2eqtxp/war/WEB-INF/lib/ojdbc7-12.1.0.2.0.jar!/oracle/jdbc/connector/OracleLocalTransaction.class

The following figure shows the JDBC library inside the application called `yyy.xxx_war`. Thus, one question is why does the application need a library that is provided by the application server?
It is possible to see the application does not define any filter to use the library from the application instead of the Weblogic library.
This can be confirmed after reading the Weblogic.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:context-root>/</wls:context-root>
<wls:security-role-assignment>
  <wls:role-name>weblogic</wls:role-name>
  <wls:principal-name>weblogic</wls:principal-name>
</wls:security-role-assignment>
</wls:weblogic-web-app>
```

In cases like this, the recommendation is to get rid of JAR files that do not make sense for the application since the application is using the one provided by the application server.
Example 2

The following case shows that even when the tag `<prefer-web-inf-classes>true</prefer-web-inf-classes>` is used there could be some problems such as:

1371620039/app:birt.war module:birt.war path:null spec-version:3.0]] Root cause of ServletException. java.lang.LinkageError: loader constraint violation in interface itable initialization: when resolving method "org.apache.xerces.dom.ElementImpl.getSchemaTypeInfo()Lorg/w3c/dom/TypeInfo;" the class loader (instance of weblogic/utils/classloaders/ChangeAwareClassLoader) of the current class, org/apache/xerces/dom/ElementImpl, and the class loader (instance of <bootloader>) for interface org/w3c/dom/Element have different Class objects for the type org/w3c/dom/TypeInfo used in the signature

According to the developer this was impossible because the application was using this

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <weblogic-web-app>
    <container-descriptor>
      <prefer-web-inf-classes>true</prefer-web-inf-classes>
    </container-descriptor>
  </weblogic-web-app>
```

With CAT it was possible to see a big number of conflicts
There are potential conflicts detected and they do not seem to have been resolved. Please review the potential solutions below.

- **355** classes are in conflict.
- Those classes are found in the following main packages:
  - `javax.wsrp.*`
  - `javax.xml.*`
  - `javax.xml.rpc.*`
  - `javax.xml.stream.*`
  - `oracle.core.*`
  - `oracle.jdbc.*`
  - `oracle.jbo.*`
  - `oracle.jbo.inter.*`
  - `oracle.jdbc.sql.*`
  - `oracle.jdbc.cmp.*`
  - `oracle.jdbc.rowset.*`
  - `oracle.xml.*`
  - `oracle.net.*`
  - `oracle.xmlrpc.*`
  - `org.apache.*`
  - `org.apache.commons.*`
Using CAT to see the conflicts, the Weblogic.xml was modified and now it looks like this:

```xml
<weblogic-web-app>
  <container-descriptor>
    <prefer-web-inf-classes>false</prefer-web-inf-classes>
    <prefer-application-packages>
      <package-name>org.apache.commons.*</package-name>
      <package-name>org.apache.xmlbeans.*</package-name>
      <package-name>org.mozilla.classfile.*</package-name>
      <package-name>org.mozilla.javascript.*</package-name>
      <package-name>org.osgi.framework.*</package-name>
      <package-name>org.osgi.resource.*</package-name>
      <package-name>org.osgi.service.*</package-name>
      <package-name>org.osgi.util.*</package-name>
      <package-name>repackage.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmlconfig.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmltools.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmllang.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmlschema.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmltools.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmllang.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmlschema.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmltools.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmllang.*</package-name>
      <package-name>schemanorg.apache.xmlbeans.system.xmlschema.*</package-name>
    </prefer-application-packages>
  </container-descriptor>
</weblogic-web-app>
```

The previous filter can be seen using CAT:

Moreover, the number of conflicts was reduced.
Therefore, in cases like this using a filter within the Weblogic.xml file is better than using the tag `<prefer-web-inf-classes>true</prefer-web-inf-classes>`

Conclusion
At least there are two ways to solve this kind of conflicts deleting JAR files that are not used by the application or filtering classes through the Weblogic.xml file where a filter is recommended as can be seen in the second case described in this document and in the bug described by Oracle in [4]

References list


