Application Server Specific Configuration Guide

This document provides app server-specific configuration information for running Apache CXF.

JBoss Application Server

JBoss Application Server (JBoss AS) comes with its own webservices stack (JBossWS) in order for providing full JavaEE support.
Starting from JBoss AS 6 M4, the default webservices stack is internally based on Apache CXF; as a consequence users might experiment classloading issues with classes from both the CXF libraries and its dependencies if included in deployments and not properly isolated. Please refer to the relevant JBoss AS documentation for details on how to turn on classloading isolation on the application server version in use.

In particular, when willing to run Apache CXF based applications on top of JBoss AS 7 series, users have basically two options:

- use JBoss AS as if it was a servlet container with no WS functionalities: this basically implies disabling the webservices subsystem for the user deployment, hence preventing the AS webservices stack from processing the ws endpoint deployment and letting the CXF libs included in the archive deal with any WS invocations when CXFServlet is hit; the webservices subsystem is turned off by adding a jboss-deployment-structure.xml as follows to the ws endpoint deployment:

  ```
  <jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
      <deployment>
          <exclude-subsystems>
              <subsystem name="webservices" />
          </exclude-subsystems>
      </deployment>
  </jboss-deployment-structure>
  ```

  this approach offers the fastest route to deploying CXF apps on JBoss AS; the drawback is that no special ws integration with JBoss AS internals is available

- rely on JBossWS integration and the Apache CXF libraries included in the application server (documentation): this implies removing any Apache CXF libs from the ws deployment as well as any other dependencies which is already included in JBoss AS (including any Java EE API jar); if included, the optional web.xml descriptor is to be rewritten according to JBossWS convention (see documentation); the Spring support is optional in JBoss AS and Spring based endpoint declaration is not the default/preferred configuration approach for ws endpoints, hence users willing to declare endpoints using Spring needs to create a org.springframework.spring module and put their endpoint declarations in a jbossws-cxf.xml descriptor; if the user application makes use of any lib besides tha JavaEE api, proper module dependencies are to be declared either using the jboss-deployment-structure.xml descriptor or the archive MANIFEST.MF (few directions on ws modules available here)

The second approach allows leveraging the full JavaEE 6 stack (including e.g. JSR-109) as well as specific ws integration with JBoss AS internals.

SpringBoot

Please see CXF SpringBoot documentation.

JAX-WS: see JAX-WS Spring Boot demo.

JAX-RS: see JAX-RS Spring Boot and JAX-RS Spring Boot Scan demos.

WebLogic

There are two ways to deploy a CXF WAR archive in WebLogic. (**Note: This has been validated on WebLogic9.2.**)

Put jars in endorsed folder

- Put the geronimo-ws-metadata_2.0_spec-1.1.1.jar in the $Weblogic_Home/jdk_../jre/lib/endorsed folder.
- Deploy the CXF war in weblogic.
  (This way is not recommended, since it might break the application server itself. The method below is preferred, as it impacts a single module only.)

Pack war in an ear, deploy the ear with weblogic-application.xml

- Create a standard J2EE application.xml file in the META-INF folder. (Take $CXF_HOME/samples/java_first_spring_support for example)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
                "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
                "http://java.sun.com/dtd/application_1_3.dtd">
<application>
        <display-name>spring_http</display-name>
        <module>
                <web>
                        <web-uri>spring_http.war</web-uri>
                        <context-root>spring</context-root>
                </web>
        </module>
</application>
```

- Create a weblogic-application.xml (Weblogic specific) in the META-INF folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-application xmlns="http://www.bea.com/ns/weblogic/90">
        <application-param>
                <param-name>webapp.encoding.default</param-name>
                <param-value>UTF-8</param-value>
        </application-param>
        <prefer-application-packages>
                <package-name>javax.jws.*</package-name>
        </prefer-application-packages>
</weblogic-application>
```

The prefer-application-packages element you see above sets up WebLogic's Filtering Classloader. Each class whose package matches one of the package-name elements listed will be searched for first within the EAR before relying on the WebLogic system classloader's version. If a package for a particular class is not listed here, WebLogic will try to load its own (possibly older) version first, so if you are getting deployment errors due to any particular class you might wish to add its package here.

Also note you can, and may need to, specify other options in the weblogic-application.xml file such as XML processing factories as shown here. See the WebLogic guide for more information.

- Run "jar cvf ..." command to create the ear and then deploy it. Alternatively, this blog entry provides a Mavenized method of building the EAR.

Websphere

For WebSphere Versions < 6.1.0.29

Adding jars to the 'endorsed' folder appears to be the main solution:

No Web Services Feature Pack for WebSphere installed

put jar in the endorsed folder

- put the wsdl4j-1.6.1.jar in the $WebSphere_HOME/java/jre/lib/endorsed folder.
- In the WebSphere console, find the specific enterprise application, click the "Class loading and update detection".
    - Mark the "Classes loaded with application class loader first" selected.
    - Mark the "Class loader for each war file in application" selected.

And then restart the Websphere server. (Because we changed the endorsed folder, we need to restart it to make it take effect).

✓ Please make sure your classpath doesn't have the servlet-2.5 library, since WebSphere6.1 is servlet-2.4 compliant!

Add your own class loader

If you put your wsdl4j-1.6.1 jar in $WAS_HOME/java/jre/lib/endorsed, all your applications will depend on your version of wsdl4j. Another solution is to create a new class loader in your server which loads before parent class loader, create a shared library with your version of wsdl4j, and add this shared library to your new class loader. This version of wsdl4j will only be available for your specific server and not affect applications running in other servers.

Step by step

1. In the WAS console navigate to **Environment > Shared Libraries**
2. Select the scope you wish your library should be visible in
3. Click **New** and set values ex: `name=MYAPP_SHARED_LIB, classpath=PATH_TO/wsdl4j-1.6.2.jar` and **Save**
4. Navigate to **Application servers > [your server name] > Java and Process Management > Class loader > New**
5. Select **Classes loaded with application class loader first** and **Save**
6. Select your new class loader and click **Shared library references**
7. Add your shared library (MYAPP_SHARED_LIB) **Save** and restart your server.

Tested in WAS 6.1 only but should work in earlier versions as well.

Another user running WS6.1 FP 23 without the web services feature pack came up with this solution that seemed to work for them:

> *Create a shared library with the following jars:*
> *jsr173_api-1.0.jar*
> *jaxp-ri-1.4.2.jar*
> *saaj-impl-1.3.2.jar*
> *wsdl4j-1.6.2.jar*
>
> *Create a new parent-first classloader and have it reference the shared library you just created. Restart everything and it should work.*

Web Services Feature Pack for WebSphere Installed

Things are way more complicated if the Web Services Feature Pack for WebSphere is installed. With this feature pack installed, it is impossible to deploy an application using CXF, because the WebSphere Web Services engine starts parsing the JAX-WS annotations of the services and tries to deploy the services.

Up to fixpack 27 (6.1.0.27) there was no possibility to disable the WebSphere Web Services engine.

For WebSphere 6.1.0.29+, V7 and V8

Follow the PDF download given within this IBM developerWorks article:http://www.ibm.com/developerworks/websphere/library/techarticles/1001_thaker/1001_thaker.html

As described in the PDF, you'll need to change the Classloader order to "Classes loaded with local class loader first (parent last)" and to disable the IBM web services engine, either for the JVM as a whole or for the particular module.

To disable for the whole JVM, set the JVM property

com.ibm.websphere.webservices.DisableIBMJAXWSEngine=true

or to disable the engine just for a specific module by adding

DisableIBMJAXWSEngine: true

to WAR/META-INF/MANIFEST.MF.

Another issue that comes up with certain versions of WebSphere is an incompatibility with the SAAJ implementation. It is recommended to use the org.apache.servicemix.bundles.saaj-impl-1.3.18_1.jar saaj impl available from http://repo1.maven.org/maven2/org/apache/servicemix/bundles/org.apache.servicemix.bundles.saaj-impl/1.3.18_1/ as that contains a recent version of SAAJ along with it's required DOM implementation which will work on the IBM JDK.

One user has reported that he was able to get CXF working on WebSphere with a minimal set of CXF jars by following the above procedures and using the list of jars:

```
FastInfoset-1.2.9.jar
aopalliance-1.0.jar
commons-logging-1.1.1.jar
cxf-2.5.2.jar
geronimo-activation_1.1_spec-1.1.jar
geronimo-annotation_1.0_spec-1.1.1.jar
geronimo-javamail_1.4_spec-1.7.1.jar
geronimo-jaxws_2.2_spec-1.1.jar
geronimo-stax-api_1.0_spec-1.0.1.jar
geronimo-ws-metadata_2.0_spec-1.1.3.jar
jars_in_war.txt
jaxb-api-2.2.3.jar
jaxb-impl-2.2.4-1.jar
neethi-3.0.1.jar
org.apache.servicemix.bundles.saaj-impl-1.3.18_1.jar
spring-aop-3.0.6.RELEASE.jar
spring-asm-3.0.6.RELEASE.jar
spring-beans-3.0.6.RELEASE.jar
spring-context-3.0.6.RELEASE.jar
spring-core-3.0.6.RELEASE.jar
spring-expression-3.0.6.RELEASE.jar
spring-web-3.0.6.RELEASE.jar
stax2-api-3.1.1.jar
woodstox-core-asl-4.1.1.jar
wsdl4j-1.6.2.jar
xmlschema-core-2.0.1.jar
```

Glassfish

CXF Interceptors will not work in Glassfish without this sun-web.xml file to configure the classloader. By default, Glassfish will use Metro for JAX-WS services so the classloader needs to be configured to allow CXF libraries to provide JAX-WS services. The following sun-web.xml xml source was added to /WEB-INF to resolve this issue:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC '-//Sun Microsystems, Inc.//DTD
Application Server 9.0 Servlet 2.5//EN'
'http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd'>
<sun-web-app>
<class-loader delegate="false"/>
</sun-web-app>
```

OC4J

This guide requires heavy customization of the OC4J configuration. Bear in mind that some of steps presented below are either undocumented or unsupported. We strongly advice you to perform those steps in a separate container, dedicated exclusively for CXF.

Also see: http://chadthedeveloper.blogspot.com/2008/06/cxf-vs-oc4j-round-1.html for other suggestions on how to configure OC4J.

Disclaimer

This guide covers only 10.1.3.X.X version of OC4J. Note that OC4J 10.1.2 is not JSE 1.5 certified server. OC4J 11_g_ is fully JEE 5.0 certified stack and comes with their own JAX-WS implementation.

Background

Oracle OC4J comes with highly customized XML stack by Oracle including SAX, StAX, JAXP, JAX-WS, SAAJ, WSDL and few others. All of those frameworks are Oracle proprietary implementations in the OC4J distribution. This gives Oracle really good interoperability between their products but it makes it rather hard to introduce something which needs different implementation of above APIs (like CXF).

OC4J 10.1.3 comes with preliminary implementation of JAX-WS (JSR-181) but this implementation is somewhat limited only to top-down scenario, with very limited customization (lack of JAXB 2.0 etc.).

Configuration overview

A few components need to be customized in OC4J to allow CFX integration:

- Xerces
- JAX-WS 2.0 APIs
- WSDL4J

Unfortunately, these components have to be configured in different parts of OC4J.

Oracle OC4J class loading

A key part of successfully integrating CXF into OC4J is to understand how class loaders work in OC4J. When starting OC4J there are generally three stages where customization could occur:

1. Virtual Machine boot
2. OC4J boot
3. CXF (application) boot

Customizing in the last step is quite easy to achieve - basically OC4J has quite powerful class loader and an easy customization console. Unfortunately there are some components that could not be configured this way. They are configured during OC4J boot. Unfortunately one of this is OC4J webservices stack (located in `$ORACLE_HOME/webservices/lib`).

Needed components

Before start please download Apache CXF 2.0.6 or better and Xerces 2.8.1

Preparing stax-api

If you use a version of CXF that includes stax-api.jar that in turn include the QName class, remove `javax.xml.namespace.QName` from the stax-api shipped with CXF. Oracle apparently has it already in `$ORACLE_HOME/j2ee/home/lib/jax-qname-namespace.jar`.

Replace the Oracle XML parser with Xerces

The basic idea behind how to do this is described in detail here

Create OC4J shared library named `cxf.foundation` and fill it with:

- xercesImpl.jar (from Xerces distribution)
- xml-apis-1.2.02.jar (from CXF-distribution)
- xalan-2.7.0.jar (ditto)
- geronimo-ws-metadata_2.0_spec-1.1.1.jar (ditto)

  When building Your application **DO NOT INCLUDE THOSE COMPONENTS** again.

Get rid of OC4J JAX-WS libraries

OC4J has preliminary support for JAX-WS, unfortunately this means that during OC4J boot it loads *outdated* JAX-WS APIs and implementation by Oracle. This occurs even before shared libraries comes into action, at a very early stage of OC4J boot. Boot-time OC4J libraries are configured in `boot.xml` file in `$ORACLE_HOME/j2ee/home/oc4j.jar` bootstrap jar. To get rid of this:

- unpack `oc4j.jar` file
- locate `META-INF/boot.xml` file and edit it
- find section

```
<!-- WS jax-rpc -->
        <code-source path="${oracle.home}/webservices/lib/jaxr-api.jar"/>
        <code-source path="${oracle.home}/webservices/lib/jaxrpc-api.jar"/>
        <code-source path="${oracle.home}/webservices/lib/jaxb-api.jar"/>
        <code-source path="${oracle.home}/webservices/lib/saaj-api.jar"/>
        <code-source path="${oracle.home}/webservices/lib/jws-api.jar" if="java.specification.version == /1\.[5-
6]/"/>
```

and comment out line which include `jws-api.jar` entry, like below

```
<!-- <code-source path="${oracle.home}/webservices/lib/jws-api.jar" if="java.specification.version == /1\.[5-6]
/"/> -->
```

- repackage `oc4j.jar` (don't forget about `MANIFEST.MF` - use `jar -m META-INF/MANIFEST.MF`)

swapping Oracle `wsdl.jar` with `wsdl4j.jar` and `jaxb.jar` API with `jaxb-api-2.0.jar`

Additionally Oracle provides it's own implementation of WSDL functionality which conflicts with `wsdl4j.jar`. To get rid of this add `-Xbootclasspath/p:"<path to wsdlj>/wsdl4j-1.6.1.jar;<path to jaxb2>/jaxb-api-2.0.jar"` option to JVM parametrs (either in command line running OC4J standalone or in OPMN).

Deploying applications

When deploying please follow those steps:

- Edit deployment plan
- Edit `Configure class loading` in the deployment plan like described here
- **Uncheck `oracle.xml`** library
- **Check `cxf.foundation`** library
- **Uncheck `Search Local Classes First`**
- do not include `xercesImpl`, `xml-apis`, `xalan` and `geronimo-ws-metadata_2.0_spec-1.1.1.jar` in `war` - those will be automatically loaded by by OC4J Shared Libraries class loader.

> You can automate above steps by packaging You `war` into `ear` archive (even though) if it's only `war` and providing `orion-application.xml` proprietary descriptor as described here. You could also provide proprietary `orion-web.xml` in Your `war` instrumenting `Search Local Classes First` attribute described above. This step is described here.

Oracle FAQ

I'm getting `java.lang.ClassCastException: org.apache.xerces.jaxp.DocumentBuilderFactoryImpl`

This primarily happens when:

- xerces is loaded twice - by shared library class loader and application class loader
- or when there is mismatch between `xerces` and `oracle` implementation of SAX API

Please be sure You properly installed and enabled for Your application `cxf.foundation` shared library as described here. If Yes please be sure that You didn't include xercesImpl.jar in Your `war`. If You still have problems please see how You can debug JAXP problems - be sure that `org.apache.xerces.jaxp.DocumentBuilderFactoryImpl` are instantiated from within JAXP and not `oracle.xml.parser.v2.DocumentBuilder`.

I cannot get WSDL (getting HTTP 500 accesing my CXF service WSDL with http://myshot/myservice?wsdl)

Please be sure that `wsdl4j.jar` is loaded before `wsdl.jar` as described here

I'm getting `java.lang.NoSuchMethodException: oracle.j2ee.ws.wsdl.extensions.soap.SOAPBodyImpl.getElementType()`

See this

I cannot get it to work still

Try something simple. Download OC4J standalone and bootstrap it from command line directly: `java [options] -jar oc4j.jar`. Enable SAX debugging. Be sure You don't include douplicated jars in Your application like `xercesImpl, xalan, xml-apis and geronimo-ws-metadata_2.0_spec-1.1.1.jar`. Review steps above once more. It works 😉 .

Integration with Application Server FAQ

1.
Q: I have this error: javax.xml.ws.WebServiceException: Cannot create SAAJ factory instance.
A: Please make sure you have the saaj-impl-1.3.jar in the classpath and make sure your app picks up this one instead of weblogic one.

Resources

Utilizing the OC4J Class Loading Framework
Deploy XFire in WebLogic
Understanding WebLogic ClassLoader
JBoss Class Configuration
Troubleshooting SAX