Artix for J2EE (JAX-WS)

Version 5.1
Oct 2008

Making Software Work Together™

Artix for J2EE (JAX-WS)

IONA Technologies

Version 5.1

Published 22 Oct 2008

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Book Details

What is Covered in This Book

This book describes how to use Artix in a J2EE application server environment. It applies to applications developed using the Artix JAX-WS API.

Who Should Read This Book

This book is aimed at J2EE application programmers who want to use the Artix JAX-WS API to develop and deploy distributed J2EE applications that are Web service enabled.

To use this guide, although you do not need an in depth knowledge of Artix concepts, WSDL and Web services, you do need to be familiar with these topics. Take a look at Using the Artix Library [http://www.iona.com/support/docs/artix/5.1/library_intro/index.htm] for pointers to books that might be of interest to you.

How to Use This Book

This book is organized into the following chapters:

- *Introduction* on page 15 gives a brief overview of the J2EE Connector Architecture and the Artix JCA Connector.
- *Exposing a J2EE application as a Web Service* on page 19 describes how to use the Artix JCA Connector to expose your J2EE application as a Web service; that is, for inbound connections.
- *Exposing a Web Service to a J2EE Application* on page 45 describes how to use the Artix JCA Connector to connect your J2EE application to a Web service; that is, for outbound connections.
- *Deploying Artix JCA Connector* on page 55 describes how to deploy Artix JCA Connector and your application to your application server.
- *Configuring Artix JCA Connector* on page 61 provides details of the activation specification properties supported by the Artix JCA Connector.

The Artix Documentation Library

For information on the organization of the Artix library, documentation conventions, and where to find additional resources, see Using the Artix Library [http://www.iona.com/support/docs/artix/5.1/library_intro/index.htm].

Introduction

Using the Artix JCA Connector, developers can easily connect their J2EE applications to Artix Web services and expose their J2EE applications as Artix Web services from within their chosen J2EE application server.

J2EE Connector Architecture Overview

Overview

The J2EE Connector Architecture (JCA) outlines a standard architecture for enabling J2EE applications to access resources in diverse Enterprise Information Systems (EISs). The goal is to standardize access to non-relational resources in the same way the JDBC API standardizes access to relational data.

The J2EE Connector Architecture is implemented in a J2EE application server and an EIS-specific resource adapter. The resource adapter plugs into the J2EE application server and provides a system library specific to, and connectivity to, that EIS.

The Artix JCA Connector is a JCA 1.5 resource adapter.

More information

For more information on the J2EE Connector Architecture, see the JCA 1.5 Specification [http://java.sun.com/j2ee/connector/download.html].

Artix JCA Connector Overview

Overview

The Artix JCA Connector is a J2EE Connector Architecture 1.5 resource adapter. It enables you to expose Artix Web services to your J2EE applications and allows you to expose your J2EE applications as Artix Web services.

The term Web services is used here to include SOAP over HTTP based services and any service that has been exposed as a Web service by Artix. The Artix JCA Connector transparently connects your J2EE applications over multiple transports to any Artix-enabled back-end service. This includes HTTP, CORBA, IIOP, IBM WebSphere MQ, Java Messaging Service (JMS), BEA Tuxedo, and TIBCO Rendezvous.

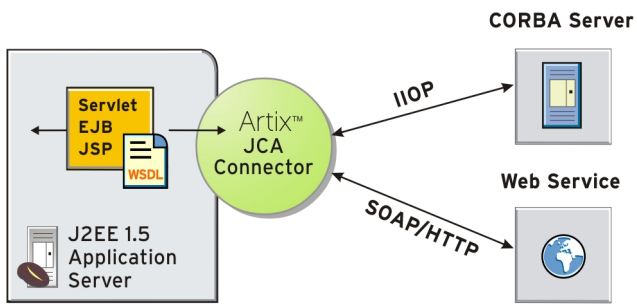
Note

To use the Artix JCA Connector your application server must support JCA 1.5 and EJB 2.1 or higher.

Graphical representation

Figure 1 on page 17 illustrates, at a high-level, how the Artix JCA Connector exposes a Web service to a J2EE application. It acts as a bridge between J2EE and SOAP over HTTP Web services. This is the simplest example. It also illustrates that the Artix JCA Connector can be used as a bridge between J2EE and a CORBA server that has been exposed as a Web service by Artix.

Figure 1. Connecting J2EE Applications to Web services



The Artix JCA Connector also enables inbound connections, allowing you to expose your J2EE application as a Web service.

Artix JCA Connector RAR file

The Artix JCA Connector resource adapter is packaged as a standard J2EE Connector Architecture resource adapter archive (RAR) file and is called `cxf.rar`. The `cxf.rar` file contains all of the classes that the Artix JCA Connector needs to manage both inbound and outbound connections.

Artix JCA Connector deployment descriptor

The Artix JCA Connector deployment descriptor file, `ra.xml`, contains information about Artix JCA Connector's resource implementation, configuration properties, transaction and security support. It describes the capabilities of the resource adapter and provides a deployer with enough information to properly configure the resource adapter in an application server environment.

An application server relies on the information in the deployment descriptor to know how to interact properly with the resource adapter. The deployment descriptor is packaged in the Artix JCA Connector RAR file.

Connection management

For information on how to use the Artix JCA Connector to manage inbound connections, see *Exposing a J2EE application as a Web Service* on page 19.

For information on how to use the Artix JCA Connector to manage outbound connections, see *Exposing a Web Service to a J2EE Application* on page 45.

Exposing a J2EE application as a Web Service

This chapter describes how to use the Artix JCA Connector for inbound connections.

Introduction

Overview

The Artix JCA Connector's inbound support makes use of the JCA 1.5 specification's message inflow contract and EJB 2.1 or higher message-driven beans (MDBs). The JCA 1.5 specification defines a framework that allows the Artix JCA Connector to be notified when a MDB starts. The Artix JCA Connector then activates the CXF service endpoint facade, which receives client requests and invokes on the MDB's listener interface.

The instructions in this chapter assume that you are familiar with writing EJBs, including Message Driven Beans and Stateless Session Beans.

Note

To use the Artix JCA Connector your application server must support JCA 1.5 and EJB 2.1 or higher.

More information

For more information about the JCA 1.5 message inflow contract, see *Chapter 12, Message Inflow* of the JCA 1.5 Specification [http://java.sun.com/j2ee/connector/download.html].

In addition, if you are interested in knowing more about what goes on behind the scenes when a resource adapter, such as the Artix JCA Connector, invokes an application asynchronously through a MDB, see JCA 1.5, Part 3: Message Inflow [http://www.ibm.com/developerworks/java/library/j-jca3/].

Usage scenarios

You can use the Artix JCA Connector to expose your J2EE application as a Web service using any of the following scenarios:

- Java first, where you implement your service as one of the following:

  a. Message Driven Bean (MDB). In this case, incoming requests do not need to be dispatched to another EJB; the MDB includes the service implementation.

     See Service Implemented as a Message Driven Bean on page 22 for more details.

  b. Stateless Session Bean (SLSB). In this case, you use an Artix-provided generic MDB to dispatch incoming requests to your SLSB.

See Service Implemented as a Stateless Session Bean on page 28 for more details.

- WSDL first, where your starting point is the service WSDL file. You use Artix to generate JAX-WS compliant Java from the WSDL file and implement your service as a SLSB. Here, again, you use the Artix-provided generic MDB to dispatch incoming requests to your SLSB.

  See WSDL First—Service Implemented as a SLSB on page 34 for more details.

The rest of this chapter describes these scenarios in more detail.

Service Implemented as a Message Driven Bean

Overview

In this scenario you implement your service as a MDB. When it starts, the MDB notifies the Artix JCA Connector. The Artix JCA Connector activates the CXF service endpoint facade, which receives client requests and invokes directly on the MDB. Incoming invocations do not have to be dispatched to another EJB.

In addition, there is no need for a service WSDL file. Artix uses the service endpoint interface to build a service model as it is defined in the activation specification `serviceInterfaceClass` property in your application's deployment descriptor file, `ejb-jar.xml`.

Advantages

The advantages of using this approach is that it preforms faster than either of the SLSB scenarios because the MDB does not need to dispatch incoming requests to another EJB.

In addition, you do not need to implement EJB Home, Local or Remote interfaces.

Disadvantages

The disadvantage of this approach is that the service endpoint interface has to be exposed as the `messagelistener-type` element in the Artix JCA Connector's deployment descriptor. This means that you have to edit the Artix JCA Connector's deployment descriptor file.

Sample application

Artix includes a working example of this scenario. You can find it in the following directory of your Artix installation:

InstallDir`/java/samples/integration/jca/inbound-mdb`

If you want to build and run this sample, please follow the instructions outlined in the `README.txt` file located in this directory. The example code shown in this section is taken from this sample application.

High-level Implementation Steps

Complete the following steps if you want to use the Artix JCA Connector to expose your J2EE application, implemented as a MDB, as a Web service:

1. Write a MDB that implements the service that you want to expose. See, for instance, `Greeter.java` located in

ArtixInstallDir/java/samples/integration/jca/inbound-mdb/src/demo/ejb

and shown in Example 1 on page 23.

Example 1. Message Driven Bean—GreeterBean.java

```
package demo.ejb;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;

import org.apache.hello_world_soap_http.Greeter;

public class GreeterBean implements MessageDrivenBean, Greeter {

      public String sayHi() {
           System.out.println("sayHi called ");
           return "Hi there!";
      }

      public String greetMe(String user) {
           System.out.println("greetMe called user = " + user);
           return "Hello " + user;
      }

      //---------------- EJB Methods
      public void ejbCreate() {
       }

      public void ejbRemove() {
      }

      public void setMessageDrivenContext(MessageDrivenContext mdc) {
      }
}
```

2. Write a deployment descriptor for your MDB. See, for instance, the `ejb-jar.xml` file located in

   ArtixInstallDir/java/samples/integration/jca/inbound-mdb/etc

   and shown in Example 2 on page 23.

Example 2. Message Driven Bean Deployment Descriptor—ejb-jar.xml

```
<?xml version="1.0"?>
...
```

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
   version="2.1">

   <enterprise-beans>
      <message-driven>
         <ejb-name>Greeter MDB</ejb-name>
         <ejb-class>demo.ejb.GreeterBean</ejb-class>
         <messaging-type>
            org.apache.hello_world_soap_http.Greeter
         </messaging-type>
         <transaction-type>Bean</transaction-type>

         <activation-config>
            <!-- displayName -->
            <activation-config-property>
               <activation-config-property-name>
                  displayName
               </activation-config-property-name>
               <activation-config-property-value>
                  MyCxfEndpoint
               </activation-config-property-value>
             </activation-config-property>

             <!-- service endpoint interface -->
             <activation-config-property>
                <activation-config-property-name>
                   serviceInterfaceClass
                </activation-config-property-name>
                <activation-config-property-value>
                   org.apache.hello_world_soap_http.Greeter
                </activation-config-property-value>
             </activation-config-property>

             <!-- address -->
             <activation-config-property>
                <activation-config-property-name>
                   address
                </activation-config-property-name>
                <activation-config-property-value>
                   http://localhost:9999/GreeterBean
                </activation-config-property-value>
              </activation-config-property>

            </activation-config>
         </message-driven>
      </enterprise-beans>
```

```
          <assembly-descriptor>
          <method-permission>
             <unchecked/>
             <method>
                <ejb-name>GreeterBean</ejb-name>
                <method-name>*</method-name>
             </method>
          </method-permission>
              <container-transaction>
                  <description/>
                  <method>
                     <description/>
                     <ejb-name>GreeterBean</ejb-name>
                     <method-name>*</method-name>
                  </method>
                  <trans-attribute>Supports</trans-attribute>
               </container-transaction>
           </assembly-descriptor>
</ejb-jar>
```

For more information about the supported activation configuration properties, see Inbound Activation Configuration on page 62.

If you are using EJB 3.0, the only change you need to make to the deployment descriptor is in the opening <ejb-jar> element. For EJB 3.0 it should read as shown in Example 3 on page 25.

Example 3. EJB 3.0 Deployment Descriptor

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

     http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
     version="3.0">
```

3. Package you application in an EJB JAR file.
4. Make a copy of the Artix JCA Connector's deployment descriptor file, `ra.xml`, which is located in the following directory of your Artix installation:

   InstallDir`/java/samples/integration/jca/inbound-mdb/etc`

5. Edit the `ra.xml` file so that the `messagelistener-type` element defines the same interface as the `messaging-type` element defined in your MDB deployment descriptor. This ensures that the Artix JCA Connector is notified when the MDB starts.

6. Build the Artix JCA Connector RAR file. It must have the following structure and contents:

Table 1. RAR File Structure & Contents: Service Implemented as MDB

Directory	Contents
META-INF	The `ra.xml` file that you modified.
Root	All of the JARs in the *`ArtixInstallDir`*`/lib` directory, except the `artix.jar`, `*manifest*.jar`s, and `*-jbi-*.jar`s. The Artix `licenses.txt` file, which is located in *`ArtixInstallDir`*`/etc`.

The sample application `build.xml` file includes a `generate.rar` target that you can use to build the RAR file (see Example 4 on page 26).

Example 4. generate.rar Target

```
<target name="generate.rar" depends="init">
    <copy file="${basedir}/etc/ra.xml" todir="${build.classes.dir}/cxf-rar/META-INF"/>
    <copy file="${cxf.home}/../etc/licenses.txt" todir="${build.classes.dir}/cxf-rar"
    failonerror="no"/>

    <copy todir="${build.classes.dir}/cxf-rar">
           <fileset dir="${cxf.home}/lib">
                    <include name="*.jar"/>
                    <exclude name="artix.jar"/>
                    <exclude name="*manifest*.jar"/>
                    <exclude name="*-jbi-*.jar"/>
            </fileset>
    </copy>
    <jar destfile="${build.classes.dir}/lib/cxf.rar"
              basedir="${build.classes.dir}/cxf-rar"/>
</target>
```

The `cxf.home` variable must be set to the *`ArtixInstallDir`*`/java` directory. This is done for you when you set your Artix environment (see

the Getting Started chapter in the Configuring and Deploying Artix Solutions, Java Runtime [http://www.iona.com/support/docs/artix/5.1/deploy/java/index.htm] guide).

7. Deploy the Artix JCA Connector RAR file and your EJB JAR file to your J2EE application server. For details, see *Deploying Artix JCA Connector* on page 55.

Service Implemented as a Stateless Session Bean

Overview

In this scenario you implement your service as a Stateless Session Bean (SLSB). Artix provides a generic MDB implementation that notifies the Artix JCA Connector when it starts. The Artix JCA Connector then activates the CXF service endpoint facade, which dispatches client requests to the generic MDB. The MDB dispatches incoming requests to your SLSB, useing the SLSB's EJB local reference (as implemented in its Local Home interface).

Advantages

The advantage of this approach is that you do not have to edit the Artix JCA Connector deployment descriptor.

In addition, there is no need for a service WSDL file. Artix uses the service endpoint interface to build a service model as it is defined in the activation specification `serviceInterfaceClass` property in your application's deployment descriptor file, `ejb-jar.xml`.

Disadvantages

The disadvantage of this approach is that it may not preform as fast as the approach described in Service Implemented as a Message Driven Bean on page 22.

Sample application

Artix includes a working example of this scenario. You can find it in the following directory of your Artix installation:

InstallDir`/java/samples/integration/jca/inbound-mdb-dispatch`

If you want to build and run this sample, please follow the instructions outlined in the `README.txt` file located in this directory. The example code shown in this section is taken from this sample application.

High-level Implementation Steps

Complete the following steps if you want to use the Artix JCA Connector to expose your J2EE application, implemented as a SLSB, as a Web service:

1. Write a SLSB that implements the service that you want to expose. See, for instance, `GreeterBean.java` located in *ArtixInstallDir*`/java/samples/integration/jca/inbound-mdb-dispatch/src/demo/ejb` and shown in Example 5 on page 29.

Example 5. Stateless Session Bean—GreeterBean.java

```
package demo.ejb;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class GreeterBean implements SessionBean {

      //------------- Business Methods
      public String sayHi() {
            System.out.println("sayHi invoked");
            return "Hi from an EJB";
      }

      public String greetMe(String user) {
            System.out.println("greetMe invoked user:" + user);
            return "Hi " + user + " from an EJB";
      }

      //------------- EJB Methods
      public void ejbActivate() {
      }

      public void ejbRemove() {
      }

      public void ejbPassivate() {
      }

      public void ejbCreate() throws CreateException {
      }

      public void setSessionContext(SessionContext con) {
      }
}
```

2. Write an EJB Local Home interface for your SLSB. See, for instance, `GreeterLocalHome.java` located in *`ArtixInstallDir`*`/java/samples/integration/jca/inbound-mdb-dispatch/src/demo/ejb` and shown in Example 6 on page 30.

Example 6. GreeterLocalHome.java

```
package demo.ejb;

import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;

public interface GreeterLocalHome extends EJBLocalHome {
     GreeterLocal create() throws CreateException;
}
```

3. Write a deployment descriptor for your SLSB and ensure that it includes:

   a. A `message-driven` element under `enterprise-beans` that references to the generic MDB as follows:

   - `ejb-class` is

     `org.apache.cxf.jca.inbound.DispatchMDBMessageListenerImpl`

   - `messaging-type` is

     `org.apache.cxf.jca.inbound.DispatchMDBMessageListener`

   b. An `ejb-local-ref` element, which is required by the MDB so it can look up the local EJB object reference for your SLSB.

   See, for instance, the `ejb-jar.xml` located in

   ArtixInstallDir`/java/samples/integration/jca/inbound-mdb-dispatch/etc`

   and shown in Example 7 on page 30.

Example 7. Stateless Session Bean Deployment Descriptor—ejb-jar.xml

```
<?xml version="1.0"?>
                            ...
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
       http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
       version="2.1">

   <enterprise-beans>
      <session>
        <ejb-name>DispatchedGreeterBean</ejb-name>
```

```
      <home>demo.ejb.GreeterHome</home>
      <remote>demo.ejb.GreeterRemote</remote>
      <local-home>demo.ejb.GreeterLocalHome</local-home>
      <local>demo.ejb.GreeterLocal</local>
      <ejb-class>demo.ejb.GreeterBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>

    <message-driven>
      <ejb-name>GreeterEndpointActivator</ejb-name>
      <ejb-class>org.apache.cxf.jca.inbound.DispatchMDBMessageListenerImpl</ejb-class>
      <messaging-type>org.apache.cxf.jca.inbound.DispatchMDBMessageListener
      </messaging-type>
      <transaction-type>Bean</transaction-type>

      <activation-config>
           <!-- display name-->
           <activation-config-property>
               <activation-config-property-name>
                  DisplayName
               </activation-config-property-name>
               <activation-config-property-value>
                  DispatchedGreeterEndpoint
               </activation-config-property-value>
           </activation-config-property>
           <!-- service endpoint interface -->
           <activation-config-property>
               <activation-config-property-name>
                  serviceInterfaceClass
               </activation-config-property-name>
               <activation-config-property-value>
                  org.apache.hello_world_soap_http.Greeter
               </activation-config-property-value>
           </activation-config-property>
           <!-- address -->
           <activation-config-property>
               <activation-config-property-name>
                  address
               </activation-config-property-name>
               <activation-config-property-value>
                  http://localhost:9999/GreeterBean
               </activation-config-property-value>
           </activation-config-property>
           <!-- targetBeanJndiName -->
           <activation-config-property>
               <activation-config-property-name>
                   targetBeanJndiName
              </activation-config-property-name>
```

```
                <activation-config-property-value>
                    java:comp/env/DispatchedGreeterLocalHome
                </activation-config-property-value>
            </activation-config-property>
        </activation-config>

        <ejb-local-ref>
            <ejb-ref-name>DispatchedGreeterLocalHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <local-home>demo.ejb.GreeterLocalHome</local-home>
            <local>demo.ejb.GreeterLocal</local>
            <ejb-link>DispatchedGreeterBean</ejb-link>
        </ejb-local-ref>
      </message-driven>

    </enterprise-beans>
</ejb-jar>
```

For more information about the supported activation configuration properties, see Inbound Activation Configuration on page 62.

If you are using EJB 3.0, the only change you need to make to the deployment descriptor is in the opening <ejb-jar> element. For EJB 3.0 it should read as shown in Example 3 on page 25.

4. Package your application in an EJB JAR file.
5. Build the Artix JCA Connector RAR file. It must have the following structure and contents:

Table 2. RAR File Structure & Contents: Service Implemented as SLSB

Directory	Contents
META-INF	The `ra.xml`, located in: *`ArtixInstallDir`*`/java/samples/integration/jca/inbound-mdb-dispatch/etc`
Root	All of the JARs in the *`ArtixInstallDir`*`/lib` directory, except the `artix.jar`, `*manifest*.jar`s, and `*-jbi-*.jar`s. The Artix `licenses.txt` file, which is located in *`ArtixInstallDir`*`/etc`.

The sample application `build.xml` file includes a `generate.rar` target that you can use to build the RAR file (see Example 4 on page 26).

Note that the `ra.xml` file `activation spec` is set to `org.apache.cxf.jca.inbound.DispatchMDBActivationSpec`, which includes a `targetBeanJndiName` configuration property that enables you to specify your SLSB's JNDI name.

6. Deploy the Artix JCA Connector RAR file and your EJB JAR file to your J2EE application server. For details, see *Deploying Artix JCA Connector* on page 55.

WSDL First—Service Implemented as a SLSB

Overview

In this scenario your service is defined in a WSDL file. You use the `artix wsdl2java` utility to generate starting point JAX-WS compliant Java code from which you implement your service as a Stateless Session Bean (SLSB).

It is similar to the scenario described in Service Implemented as a Stateless Session Bean on page 28. Again you make use of the generic MDB implementation provided by Artix. It notifies the Artix JCA Connector when it starts and the Artix JCA Connector then activates the CXF service endpoint facade. The service endpoint facade dispatches client requests to the generic MDB. The MDB performs a JNDI lookup to obtain a reference to your SLSB and dispatches incoming requests to it.

Differences between the WSDL-first SSLB and Java-first SLSB

The primary differences between this approach and the approach described in Service Implemented as a Stateless Session Bean on page 28 is that:

- You can configure the Artix bus directly by including a `cxf.xml` Artix Java configuration file in your EJB JAR file.
- Artix creates a service bean based on the service WSDL file and you must include the WSDL file in the EJB JAR file.
- Your EJB deployment descriptor must contain additional activation configuration properties, including:
  - `busConfigLocation`, which points to the location of the Artix Java configuration file.
  - `wsdlLocation`, which points to the location of the service WSDL file.
  - `endpointName`, which points to the `PortType` QName in the WSDL file.
  - `serviceName`, which points to the `Service Name` QName in the WSDL file.

For more information on activation configuration properties, see Inbound Activation Configuration on page 62.

Advantages

One advantage of using this approach is the ability to configure directly the Artix bus.

Sample application

Artix includes a working example of this scenario. You can find it in the following directory of your Artix installation:

`InstallDir/java/samples/integration/jca/inbound-mdb-dispatch-wsdl`.

If you want to build and run this sample, please follow the instructions outlined in the `README.txt` file located in this directory. The example code shown in this section is taken from this sample application.

Implementation steps

Complete the following steps if you want to use the Artix JCA Connector to expose your J2EE application, defined in a WSDL file and implemented as a SLSB, as a Web service:

1. Set your Artix environment using the `artix_java_env` script, which is located in the `ArtixInstallDir/java/bin` directory.

   For more information on the `artix_java_env` script, see the Getting Started chapter in the Configuring and Deploying Artix Solutions, Java Runtime [http://www.iona.com/support/docs/artix/5.1/deploy/java/index.htm] guide.

2. Obtain a copy of, or details of the location of, the WSDL file that defines the Web service that your application will implement.

   This step assumes that the Web service WSDL file already exists. If, however, you need to develop a WSDL file, please refer to the Writing Artix Contracts [http://www.iona.com/support/docs/artix/5.1/contract/index.html] guide.

3. Map the WSDL file to Java to obtain starting point JAX-WS compliant Java code. Artix provides an `artix wsdl2java` command-line utility that does this for you. To generate JAX-WS compliant Java code from your WSDL file, run the following command:

```
artix wsdl2java -d [output-directory] -p [wsdl-namespace=]
PackageName wsdlfile
```

The `artix wsdl2java`parameters are defined as follows:

Table 3. artix wsdl2java Parameters

-d [output-directory]	Specifies the directory to which the generated code is written. The default is the current working directory.
-p [wsdl-namespace=] PackageName	Specifies the name of the Java package to use for the generated code. You can optionally map a WSDL namespace to a particular package name if your contract has more than one namespace.
wsdlfile	Specifies the WSDL file from which the Java code is being generated.

For more information on the `artix wsdl2java` command-line utility, see the "Generating Code from WSDL" chapter in the Artix Command Line Reference [http://www.iona.com/support/docs/artix/5.1/command_ref/index.html].

4. Write a stateless session bean (SLSB) that implements the service that you want to expose. See, for instance, `GreeterBean.java` located in

```
ArtixInstallDir/java/samples/integration/jca/inbound-mdb-
dispatch-wsdl/src/demo/ejb
```

and shown in Example 8 on page 36.

Example 8. WSDL First SLSB—GreeterBean.java

```
package demo.ejb;

import java.util.logging.Logger;
import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.types.FaultDetail;
```

```
public class GreeterBean implements SessionBean, Greeter {

      private static final Logger LOG =
            Logger.getLogger(GreeterBean.class.getPackage().getName());

       //------------- Business Methods
       // (copied from wsdl_first sample)

       public String greetMe(String me) {
             LOG.info("Executing operation greetMe");
             System.out.println("Executing operation greetMe");
             System.out.println("Message received: " + me + "\n");
             return "Hello " + me;
       }

       public void greetMeOneWay(String me) {
             LOG.info("Executing operation greetMeOneWay");
             System.out.println("Executing operation greetMeOneWay\n");
             System.out.println("Hello there " + me);
       }

       public String sayHi() {
             LOG.info("Executing operation sayHi");
             System.out.println("Executing operation sayHi\n");
             return "Bonjour";
       }

       public void pingMe() throws PingMeFault {
             FaultDetail faultDetail = new FaultDetail();
             faultDetail.setMajor((short)2);
             faultDetail.setMinor((short)1);
             LOG.info("Executing operation pingMe, throwing PingMeFault exception");
             System.out.println("Executing operation pingMe, throwing PingMeFault
             exception\n");
             throw new PingMeFault("PingMeFault raised by server", faultDetail);
       }

       //------------- EJB Methods
       public void ejbActivate() {
       }

       public void ejbRemove() {
       }

       public void ejbPassivate() {
       }

       public void ejbCreate() throws CreateException {
       }
```

```
        public void setSessionContext(SessionContext con) {
        }

}
```

5. Write an EJB Local Home interface for your SLSB. See, for instance, `GreeterLocalHome.java` located in

```
ArtixInstallDir/java/samples/integration/jca/
inbound-mdb-dispatch-wsdl/src/demo/ejb
```

and shown in Example 9 on page 38.

Example 9. WSDL First—GreeterLocalHome.java

```
package demo.ejb;

import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;

public interface GreeterLocalHome extends EJBLocalHome {
GreeterLocal create() throws CreateException;
}
```

6. Write an Artix Java configuration file if you want to configure the Artix bus directly. See, for instance, the `cxf.xml` Artix Java configuration file located in

```
ArtixInstallDir/java/samples/integration/jca/
inbound-mdb-dispatch-wsdl/etc
```

and shown in Example 10 on page 39. It shows how you configure logging.

For more information on how to configure the Artix bus, see Configuring and Deploying Artix Solutions, Java Runtime [http://www.iona.com/support/docs/artix/5.1/deploy/java/index.htm].

For information on how to configure Artix security, see the Artix Security Guide [http://www.iona.com/support/docs/artix/5.1/security_guide/index.htm].

Example 10. cxf.xml—Configuring Logging

```
<beans xmlns="http://www.springframework.org/schema/beans"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:cxf="http://cxf.apache.org/core"
               xsi:schemaLocation="
               http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
               http://www.springframework.org/schema/beans
               http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
...
      <cxf:bus>
            <cxf:features>
                  <cxf:logging/>
            </cxf:features>
      </cxf:bus>
</beans>
```

7. Write a deployment descriptor for your SLSB and ensure that it includes:

   a. A `message-driven` element under `enterprise-beans` that references to the generic MDB as follows:

   - `ejb-class` is `org.apache.cxf.jca.inbound.DispatchMDBMessageListenerImpl`
   - `messaging-type` is `org.apache.cxf.jca.inbound.DispatchMDBMessageListener`

   b. An `ejb-local-ref` element, which is required by the MDB so it can look up the local EJB object reference for your SLSB.

   See, for instance, the `ejb-jar.xml` file in

   ```
   ArtixInstallDir/java/samples/integration/jca/
   inbound-mdb-dispatch-wsdl/etc
   ```

   and shown in Example 11 on page 39.

Example 11. WSDL First SLSB Deployment Descriptor—ejb-jar.xml

```
<?xml version="1.0"?>
```

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
                version="2.1">

    <enterprise-beans>
       <session>
          <ejb-name>GreeterWithWsdlBean</ejb-name>
          <local-home>demo.ejb.GreeterLocalHome</local-home>
          <local>demo.ejb.GreeterLocal</local>
          <ejb-class>demo.ejb.GreeterBean</ejb-class>
          <session-type>Stateless</session-type>
          <transaction-type>Container</transaction-type>
       </session>

       <message-driven>
          <ejb-name>GreeterEndpointActivator</ejb-name>
          <ejb-class>org.apache.cxf.jca.inbound.DispatchMDBMessageListenerImpl
          </ejb-class>
          <messaging-type>org.apache.cxf.jca.inbound.DispatchMDBMessageListener
          </messaging-type>
          <transaction-type>Bean</transaction-type>

          <activation-config>
             <!-- bus configuration location -->
             <activation-config-property>
                <activation-config-property-name>
                   busConfigLocation
               </activation-config-property-name>
               <activation-config-property-value>
                  etc/cxf.xml
               </activation-config-property-value>
            </activation-config-property>
            <!-- wsdl location -->
            <activation-config-property>
               <activation-config-property-name>
                  wsdlLocation
               </activation-config-property-name>
               <activation-config-property-value>
                  wsdl/hello_world.wsdl
               </activation-config-property-value>
            </activation-config-property>
            <!-- service name -->
            <activation-config-property>
               <activation-config-property-name>
                  serviceName
               </activation-config-property-name>
               <activation-config-property-value>
```

```
                {http://apache.org/hello_world_soap_http}SOAPService
            </activation-config-property-value>
        </activation-config-property>
        <!-- endpoint name -->
        <activation-config-property>
            <activation-config-property-name>
                endpointName
            </activation-config-property-name>
            <activation-config-property-value>
                {http://apache.org/hello_world_soap_http}SoapPort
            </activation-config-property-value>
        </activation-config-property>
        <!-- service interface class -->
        <activation-config-property>
            <activation-config-property-name>
                serviceInterfaceClass
            </activation-config-property-name>
            <activation-config-property-value>
                org.apache.hello_world_soap_http.Greeter
            </activation-config-property-value>
        </activation-config-property>
        <!-- address -->
        <activation-config-property>
            <activation-config-property-name>
                address
            </activation-config-property-name>
            <activation-config-property-value>
                http://localhost:9000/SoapContext/SoapPort
            </activation-config-property-value>
        </activation-config-property>
        <!-- display name-->
        <activation-config-property>
            <activation-config-property-name>
                displayName
            </activation-config-property-name>
            <activation-config-property-value>
                GreeterWithWsdlEndpoint
            </activation-config-property-value>
        </activation-config-property>
        <!-- targetBeanJndiName -->
        <activation-config-property>
            <activation-config-property-name>
                targetBeanJndiName
            </activation-config-property-name>
            <activation-config-property-value>
                java:comp/env/GreeterWithWsdlLocalHome
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
```

```
         <ejb-local-ref>
            <ejb-ref-name>GreeterWithWsdlLocalHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <local-home>demo.ejb.GreeterLocalHome</local-home>
            <local>demo.ejb.GreeterLocal</local>
            <ejb-link>GreeterWithWsdlBean</ejb-link>
         </ejb-local-ref>
      </message-driven>

   </enterprise-beans>
</ejb-jar>
```

The `ejb-jar.xml` file in this scenario includes additional activation configuration properties. These properties are used during endpoint activation and point to:

- The Artix Java configuration file: `busConfigLocation`
- The service WSDL file: `wsdlLocation`
- The service name QName as defined in the WSDL file: `serviceName`
- The `PortType` QName as defined in the WSDL file: `endpointName`

For more information on activation configuration properties, see *Configuring Artix JCA Connector* on page 61.

If you are using EJB 3.0, the only change you need to make to the deployment descriptor is in the opening <ejb-jar> element. For EJB 3.0 it should read as shown in Example 3 on page 25.

8. Build your EJB JAR file and remember to include the service WSDL file in a `wsdl` directory and the Artix Java configuration file, if you have one, in an `etc` directory.

9. Build the Artix JCA Connector RAR file. It must have the structure and contents shown in Table 2 on page 32.

   The sample application `build.xml` file includes a `generate.rar` target that you can use to build the RAR file (see Example 4 on page 26).

10. Deploy the Artix JCA Connector RAR file and your EJB JAR file to your J2EE application server. For details, see *Deploying Artix JCA Connector* on page 55.

Exposing a Web Service to a J2EE Application

You can use the Artix JCA Connector to connect your J2EE applications to Web services, otherwise known as outbound connections. This chapter walks you through the steps involved.

Introduction

Overview

The Artix JCA Connector includes a connection management API that allows you to get a connection from your J2EE application to an Artix Web service. The Artix JCA Connector API usage pattern is consistent with general connection management in J2EE.

Sample applications

Artix includes working samples that demonstrate how outbound connections work. You can find them in the following directories of your Artix installation:

- *ArtixInstallDir*/java/samples/integration/jca/outbound-ws-security

  and

- *ArtixInstallDir*/java/samples/integration/jca/outbound-ws-security-conf

If you want to build and run these samples, please follow the instructions outlined in the `README.txt` file located in each directory. The example code shown in this chapter is taken from these sample applications.

Implementation Steps

Steps

The following is a list of the steps that you need to complete to expose your J2EE application to a Web service using the Artix JCA Connector. It assumes that the Web service WSDL file already exists. If, however, you need to develop a WSDL file, please refer to the Writing Artix Contracts [http://www.iona.com/support/docs/artix/5.1/contract/index.html] guide.

1. Set your Artix environment (see the Getting Started chapter in the Configuring and Deploying Artix Solutions, Java Runtime [http://www.iona.com/support/docs/artix/5.1/deploy/java/index.html] guide).
2. Obtain a copy of, or details of the location of, the WSDL file that defines the Web service to which your application needs to connect.
3. Map the WSDL file to Java to obtain the Java interfaces that you will use when writing your application. Artix provides an `artix wsdl2java` command-line utility that does this for you. The Artix WSDL-to-Java mapping is based on the JAX-WS standard.

   To generate JAX-WS compliant Java from your WSDL file, run the following command:

   ```
   artix wsdl2java -d [output-directory] -p [wsdl-namespace=]
   PackageName wsdlfile
   ```

   The `artix wsdl2java`parameters are defined as shown in Table 3 on page 36.
4. Write your application. For details, see Writing Your Application on page 49.
5. Package your application. For details, see Packaging Your Application on page 52.
6. Build the Artix JCA Connector RAR file. It must have the following structure and contents:

Table 4. Outbound Connections: RAR File Structure & Contents

Directory	Contents
META-INF	The `ra.xml` file located in *`ArtixInstallDir`*`\java\samples\integration\jca\outbound-ws-security\etc`

Directory	Contents
Root	All of the JARs in the *ArtixInstallDir*/lib directory, except the artix.jar, *manifest*.jars, and *-jbi-*.jars. The Artix licenses.txt file, which is located in *ArtixInstallDir*/etc.

The sample application build.xml file includes a generate.rar target that you can use to build the RAR file (see Example 4 on page 26).

7. Deploy the Artix JCA Connector RAR file and your application to your J2EE application server. For details, see *Deploying Artix JCA Connector* on page 55.

Writing Your Application

Connection Management API Definition

The Artix JCA Connector connection management API is packaged in `org.apache.cxf.jca.outbound` and consists of two interfaces—`CXFConnectionFactory` and `CXFConnection`. It is packaged in the following `.jar` file:

ArtixInstallDir`/java/lib/cxf-integration-jca-`*Version*`-fuse.jar`

The `CXFConnectionFactory` interface provides the methods to create a `CXFConnection` that represents a Web service defined by the supplied parameters. It is the type returned from an environment naming context lookup of the Artix JCA Connector by a J2EE component and is the entry point to gaining access to a Web service.

The `CXFConnection` interface provides a handle to a connection managed by the J2EE application server. It is the super interface of the Web service proxy returned by `CXFConnectionFactory`.

Usage pattern

To use `CXFConnectionFactory` your application needs to:

1. Look up a `CXFConnectionFactory` in the application server's JNDI registry.
2. Use the `CXFConnectionFactory.getConnection` method to get a `CXFConnection`.

   The `CXFConnectionFactory.getConnection` method takes one parameter, `CXFConnectionSpec`, which the takes following fields:

   - `serviceName`: the QName of the service. This is required.
   - `endpointName`: the QName of the endpoint; i.e. the port name. This is required.
   - `wsdlURL`: the URL of the WSDL file. This is required.
   - `serviceClass`: the service interface class. This is required.

- `busConfigURL`: the URL of Artix Java bus configuration, if such configuration exists. It allows you to configure directly the Artix bus. This is optional.

  For more information on how to configure the Artix bus, see Configuring and Deploying Artix Solutions, Java Runtime [http://www.iona.com/support/docs/artix/5.1/deploy/java/index.htm].

  For information on how to configure Artix security, see the Artix Security Guide [http://www.iona.com/support/docs/artix/5.1/security_guide/index.htm].

  The `busConfigURL` setting overrides any configuration that has been set using the Artix JCA Connector `busConfigLocation` activation configuration property (See Inbound Activation Configuration on page 62 for more detail).

- `address`: the transport address. This is optional.

3. Use the `CXFConnection.getService` method to obtain a Web service client.
4. Close the `CXFConnection`.
5. Invoke on the service.

   The Web service client can still be used after the `CXFConnection` is closed.

Example of using the Connection Management API

The code shown in Example 12 on page 51 is taken from the `HelloWorldServlet.java` file, which is part of the `outbound-ws-security` sample. It shows how to use the Artix JCA Connector connection management API. It has been simplified to make it easier to explain.

The `HelloWorldServlet.java` file is located in:

```
ArtixInstallDir/java/samples/integration/jca/
outbound-ws-security/src/demo/servlet
```

Example 12. HelloWorldServlet—Outbound Connections

```
❶Context ctx = new InitialContext();
CXFConnectionFactory factory = (CXFConnectionFactory)ctx.lookup(EIS_JNDI_NAME);

❷CXFConnectionSpec spec = new CXFConnectionSpec();
    spec.setServiceClass(Greeter.class);
   spec.setServiceName(new QName("http://apache.org/hello_world_soap_http", "SOAPService"));

   spec.setEndpointName(new QName("http://apache.org/hello_world_soap_http", "SoapPort"));

    spec.setWsdlURL(getClass().getResource("/wsdl/hello_world.wsdl"));
    CXFConnection connection = null;
    try {
          connection = getConnection(spec);

          ❸Greeter greeter = connection.getService(Greeter.class);

          ❹connection.close();

          ❺greeter.sayHi();
...
         }
```

The code shown in Example 12 on page 51 can be explained as follows:

❶ Retrieve the connection factory from JNDI.

❷ Create the connection and use `CXFConnectionSpec`to specifying:

- The service class.
- A QName that identifies which service in the WSDL file to use.
- A QName that identifies which port in the WSDL file to use
- The WSDL file URL.

❸ Obtain a Web service client.

❹ Close the connection to the service and return to the application server connection pool. Remember you can close the connection and continue using the client.

❺ Invoke on the service.

Packaging Your Application

Overview

When packaging and deploying your J2EE application you must declare the resource reference used in your code in your application deployment descriptor and map that resource reference to a resource. In addition, you need to package the Web service interface classes with your application.

Declaring the resource reference

You must declare the resource reference used in your code in your application deployment descriptor, `web.xml`, by adding a `resource-ref` tag. See Example 13 on page 52.

Example 13. Declaring the resource reference

```
<resource-ref>
     <res-ref-name>eis/CXFConnectionFactory</res-ref-name>
     <res-type>org.apache.cxf.jca.outbound.CXFConnectionFactory</res-type>
     <res-auth>Container</res-auth>
</resource-ref>
```

Mapping the resource reference

You must map the resource reference used in your code to the resource. How you do this is dependent on the application server that you are using. For example, if you are using WebSphere you can use the WebSphere Administrative Console to map the resource reference to the resource while deploying the Artix JCA Connector. See *Deploying Artix JCA Connector* on page 55 and the WebSphere documentation for details.

Packaging details

When packaging your application, include the Java classes that are generated by the `artix wsdl2java` utility, any other classes that are associated with your application, and the WSDL file.

For example, the `outbound-ws-security` sample application is packaged in a WAR file as follows:

- `WEB-INF/classes`: includes the application Java class files, the Java classes that are generated from the WSDL file.
- `WEB-INF/classes/wsdl`: WSDL file.

- `WEB-INF/lib`: includes a `common.jar` file that contains the `DemoServletBase.class` file, which the sample application extends.

Please refer to the J2EE specification and your J2EE vendor documentation for more information on application packaging.

Deploying Artix JCA Connector

How you deploy the Artix JCA Connector is dependent on the J2EE application server that you are using. This chapter provides some basic deployment steps and uses WebSphere 6.1 as an example application server.

Introduction

Overview

How you deploy the Artix JCA Connector is dependent on the J2EE application server that you are using. This chapter describes how to set your Artix environment and provides some basic deployment steps for WebSphere 6.1. It assumes that you have already built the Artix JCA Connector RAR file and your application JAR file. If not, please refer to either:

- *Exposing a J2EE application as a Web Service* on page 19
- *Exposing a Web Service to a J2EE Application* on page 45

More detailed information

For more detailed information on how to deploy a JCA resource adapter, please refer to your J2EE application server documentation.

Setting your Environment

Overview

To use Artix JCA Connector with your application server, ensure that:

- The `cxf-manifest.jar`, which is located in *`ArtixInstallDir`*`/java/lib` directory, is on your `CLASSPATH`.
- That JDK and ant `bin` directories are on your `PATH`.

You do not need to, and should not, source the Artix environment before running your application server.

Deploying to WebSphere 6.1

Overview

This section provides basic information on deploying the Artix JCA Connector and your application to WebSphere 6.1. For more detailed information, please refer to your WebSphere documentation.

Prerequisites

The following prerequisites apply to WebSphere 6.1:

- Copy the `wsdl4j-1.6.1.jar` from your *`ArtixInstallDir`*`/java/lib` directory to your *`WAS_HOME`*`/AppServer/java/jre/lib/endorsed` directory. If the endorsed directory does not exist, create it.

  Restart WebSphere.

- Make sure your environment is set correctly. See Setting your Environment on page 57 for details.

Deploying the Artix JCA Connector

You must deploy the Artix JCA Connector to WebSphere before you deploy your application. In addition, please make sure that the Artix JCA Connector has not already been deployed to your application server.

To deploy the Artix JCA Connector in WebSphere 6.1 complete the following steps:

1. Logon to WebSphere Integrated Solution Console. The default address is:

   `http://`*`hostname`*`:9060/ibm/console/login.do`

2. Navigate to **Resources | Resource adapters | Resource adapters.**
3. On the Resource adapters page, click **Install RAR**.
4. On the Install RAR File page, select the **Local path** radio button if the browser that you are running is on the same machine as the WebSphere server. Otherwise, select the **Server path** radio button.
5. Specify or browse to where you have built the `cxf.rar` file and click **Next**.
6. On the next page, click **OK** to install the Resource Adapter.
7. On the next page, click the **CXF JCA Connector** link to edit the Resource Adapter.

8. On the Configuration page, click the **J2C activation specification** link.
9. On the next page, click **New** to create a new Activation Specification.
10. On the next page, enter `MyActivationSpec` in the Name textbox and click **OK**.

    The JNDI name is optional. If it is omitted, a JNDI name is created for you as `eis/<ActivationSpecName>`, where `<ActivationSpecName>` is `MyActivationSpec`.
11. Click **Save** to commit the configuration.

You can specify activation configuration values in the new activation specification you just created.

For inbound connections, the activation specification is associated with your MDB later. The MDB's deployment descriptor can define activation configuration values to override the values specified in the associated activation specification. See Inbound Activation Configuration on page 62 for more detail.

Deploying Your application

To deploy your application to WebSphere 6.1, complete the following steps. For more detail, please consult your WebSphere documentation.

1. Logon to WebSphere Integrated Solution Console. The default address is:

   `http://<hostname>:9060/ibm/console/login.do`
2. Navigate to **Applications | Install new Applications**
3. On the **Preparing for the application installation** page, select the **Local path** radio button if the browser that you are running is on the same machine as the WebSphere server. Otherwise, select the **Server path** radio button.
4. Specify or browse to the path where you have your application JAR file stored and click **Next**.
5. On the **Step 1: Select installation options** page, click **Next**.
6. On the **Step 2: Map modules to servers** page, click **Next**.

7. On the **Step 3: Bind listeners for message-driven beans** page, in the far right column, click the **Activation Specification** radio button.
8. Specify the Target Resource JNDI Name as below and click **Next**.

   ```
   eis/MyActivationSpec
   ```

9. On the **Step 4: Summary** page, click **Finish**.
10. Click the **Save** link to commit the configuration.
11. Navigate to **Applications | Enterprise Applications**.
12. Select the box next to your application JAR file and click **Start** to start the MDB.

Configuring Artix JCA Connector

Inbound Activation Configuration

Introduction

Activation specifications are part of the configuration of inbound messaging support provided by a JCA 1.5 resource adapter, such as Artix JCA Connector. Resource adapters that support inbound messaging define one or more types of message listener in their deployment descriptors. This is defined in the `messagelistener` element in the `ra.xml` file. The message listener is the interface that the resource adapter uses to communicate inbound messages to the message endpoint. For each type of message listener that a resource adapter implements, the resource adapter defines an associated activation specification, which defines configuration properties for the receiving endpoint.

The Artix JCA Connector inbound support includes two types of message listener and two activation specification classes, one for each message listener type.

Table 5. Message Listeners and Activation Specifications

Message Listener Type	Activation Specification Class	Supported Properties
Target service interface, used when MDB also implements the target service. See Service Implemented as a Message Driven Bean on page 22 for an example use case.	`org.apache.cxf.jca.inbound.MDBActivationSpec`	See Table 6 on page 63.

`org.apache.cxf.jca.inbound.`
 	
`DispatchMDBMessageListener`
 	
See Service Implemented as a Stateless Session Bean on page 28for an example use case.
	`org.apache.cxf.jca.inbound.DispatchMDBActivationSpec`	See Table 6 on page 63 and Table 7 on page 63.

Supported Properties

Table 6 on page 63 shows the activation configuration properties that are supported when the target service interface is specified as the message listener type and `org.apache.cxf.jca.inbound.MDBActivationSpec` is specified as the activation specification class in the Artix JCA Connector `ra.xml` file.

Table 6. Service Implemented as MDB: Supported Activation Configuration Properties

Property Name	Required	Description
wsdlLocation	No	A string that specifies the location of the Web service WSDL file.
schemaLocations	No	String that specifies the schema locations, each one separated by a comma.
serviceInterfaceClass	Yes	String that specifies the service interface class name.
busConfigLocation	No	String that specifies the location of any Artix Java bus configuration files.
address	No (if specified in WSDL file)	String the specifies the transport address.
endpointName	Yes	String that specifies the `PortType` QName in the WSDL file.
serviceName	Yes	String that specifies the service name QName in the WSDL file.
displayName	Yes	String that specifies the name used for logging and as a key in a map of endpoints.

Table 6 on page 63 and Table 7 on page 63 show the activation configuration properties that are supported when `org.apache.cxf.jca.inbound.DispatchMDBMessageListener` is specified as the message listener type and `org.apache.cxf.jca.inbound.DispatchMDBActivationSpec` is specified as the activation specification class in the Artix JCA Connector `ra.xml` file.

Table 7. Service Implemented as a SLSB: Supported Activation Configuration Properties

Property Name	Required	Description
targetBeanJndiName	Yes	A string that specifies the JNDI name of the target session bean.

Setting activation configuration properties

Activation configuration properties can be set in any of the following:

a. The application deployment descriptor.

b. Activation specification, which can be set when deploying Artix JCA Connector.

c. The Artix JCA Connector deployment descriptor, `ra.xml`.

Values specified in the `ejb-jar.xml` file override those set in the activation specification and the `ra.xml` file. Values specified in the activation specification override those set in the `ra.xml` file.

Examples of setting

For an example of how the activation configuration properties are set, see:

a. The `ra.xml` located in *ArtixInstallDir*`/java/samples/integration/jca/inbound-mdb-dispatch/etc`, the relevant sections of which are shown in Example 14 on page 64.

b. The `ejb-jar.xml` file located in *ArtixInstallDir*`/java/samples/integration/jca`, the relevant sections of which are shown in Example 15 on page 64.

Example 14. Activation Specification in ra.xml

```
<messagelistener>
      <messagelistener-type>
            org.apache.cxf.jca.inbound.DispatchMDBMessageListener
      </messagelistener-type>
      <activationspec>
            <activationspec-class>
                  org.apache.cxf.jca.inbound.DispatchMDBActivationSpec
            </activationspec-class>
            <required-config-property>
                <config-property-name>displayName
                </config-property-name>
            </required-config-property>
            <required-config-property>
                <config-property-name>targetBeanJndiName
                </config-property-name>
            </required-config-property>
        </activationspec>
</messagelistener>
```

Example 15. Activation Specification in ejb-jar.xml

```
<activation-config>
    <!-- display name-->
    <activation-config-property>
```

```
      <activation-config-property-name>
         DisplayName
      </activation-config-property-name>
      <activation-config-property-value>
         DispatchedGreeterEndpoint
      </activation-config-property-value>
      </activation-config-property>
   <!-- service endpoint interface -->
   <activation-config-property>
         <activation-config-property-name>
            serviceInterfaceClass
         </activation-config-property-name>
         <activation-config-property-value>
            org.apache.hello_world_soap_http.Greeter
         </activation-config-property-value>
   </activation-config-property>
   <!-- address -->
   <activation-config-property>
      <activation-config-property-name>
         address
      </activation-config-property-name>
      <activation-config-property-value>
         http://localhost:9999/GreeterBean
      </activation-config-property-value>
   </activation-config-property>
   <!-- targetBeanJndiName -->
   <activation-config-property>
      <activation-config-property-name>
         targetBeanJndiName
      </activation-config-property-name>
      <activation-config-property-value>
         java:comp/env/DispatchedGreeterLocalHome
      </activation-config-property-value>
   </activation-config-property>
</activation-config>
```

Index

T