Artix™ ESB

Configuring and Deploying Artix Solutions, Java Runtime

Version 5.0, July 2007

Making Software Work Together™

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logos, Orbix, Artix, Making Software Work Together, Adaptive Runtime Technology, Orbacus, IONA University, and IONA XMLBus are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this book. This publication and features described herein are subject to change without notice.

Copyright © 2001-2007 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

Updated: October 10, 2007

Contents

List of Figures

List of Tables

Preface

What is Covered in this Book

Configuring and Deploying Artix Solutions, Java Runtime explains how to configure and deploy Artix Java services and applications, including those written in JAX-WS and JavaScript. For details of using Artix in a C++ or JAX-RPC environment, see Configuring and Deploying Artix Solutions, C++ Runtime.

Who Should Read this Book

The main audience of *Configuring and Deploying Artix Solutions, Java Runtime* is Artix system administrators. However, anyone involved in designing a large-scale Artix solution will find this book useful.

Knowledge of specific middleware or messaging transports is not required to understand the general topics discussed in this book. However, if you are using this book as a guide to deploying runtime systems, you should have a working knowledge of the middleware transports that you intend to use in your Artix solutions

Organization of this Guide

This guide is divided into the following chapters:

- Chapter 1, Getting Started, which describes how to set up your Artix Java environment.
- Chapter 2, Artix Java Configuration, which describes Artix Java configuration.
- Chapter 3, Deploying to the Spring Container, which describes how to deploy an Artix Java endpoint to the Spring container.
- Chapter 4, Deploying to a Servlet Container, which describes how to deploy an Artix Java endpoint to a servlet container.
- Chapter 5, Artix Logging, which describes how to use logging.
- Chapter 6, Enabling Reliable Messaging, which describes how to enable and configure Web Services Reliable Messaging (WS-RM).
- Chapter 7, Publishing WSDL Contracts, which describes how to enable the Artix Java WSDL publishing service.
- Chapter 8, Enabling High Availability, which describes how to enable and configure both static failover and dynamic failover.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see Using the Artix Library.

CHAPTER 1

Getting Started

This chapter explains how to set your Artix Java runtime system environment.

In this chapter

This chapter discusses the following topics:

Setting your Artix Java Environment

Overview

To use the Artix design tools and runtime environment, the host computer must have several IONA-specific environment variables set. These variables can be configured during installation, or later using the `artix_java_env` script, or configured manually.

Running the artix_java_env script

The Artix installation process creates a script named `artix_java_env`, which captures the information required to set your host's environment variables. Running this script configures your system to use the Artix Java runtime. The script is located in the following directory of your Artix installation:

```
ArtixInstallDir\java\bin\artix_java_env
```

Artix Java Environment Variables

Overview

This section describes the following environment variables in more detail:

- ARTIX_JAVA_HOME
- JAVA_HOME
- IT_ARTIX_BASE_DIR
- ANT_HOME
- ACTIVEMQ_HOME
- ACTIVEMQ_VERSION
- SPRING_CONTAINER_HOME
- IT_WSDLGEN_CONFIG_FILE
- PATH

Note: You do not have to manually set your environment variables. You can configure them during installation, or set them later by running the provided `artix_java_env` script.

The environment variables are explained in Table 1:

Table 1: *Artix Java Environment Variables*

Variable	Description
ARTIX_JAVA_HOME	ARTIX_JAVA_HOME points to the top level of your Artix Java installation. For example, on Windows, if you install Artix into the C:\Program Files\IONA directory, ARTIX_JAVA_HOME should be set to: C:\Program Files\IONA\java
JAVA_HOME	JAVA_HOME specifies the directory path to your system's JDK. This must be set to use the Artix Designer GUI. This defaults to the JVM selected when installing Artix. The Artix installer, by default, installs a JRE. It also, however, allows you to specify a previously installed JVM.

Table 1: *Artix Java Environment Variables*

Variable	Description
IT_ARTIX_BASE_DIR	IT_ARTIX_BASE_DIR points to the top level of your Artix installation. For example, on Windows, if you install Artix into the C:\Program Files\IONA directory, IT_ARTIX_BASE_DIR should be set to that directory.
ANT_HOME	ANT_HOME specifies the directory path to the ant utility installed by Artix. The default location is: IT_ARTIX_BASE_DIR\tools\ant The ant utility is a Java-based build tool. The build.xml files located in the sample directories contain the instructions for building the sample applications, in an XML format that is understood by the ant utility. ANT_HOME must be set to allow the building and running of the Artix Java samples. For more information about ant, see http://ant.apache.org/
ACTIVEMQ_HOME	ACTIVEMQ_HOME specifies the directory path to the ActiveMQ message broker installed by Artix. The default location is: IT_ARTIX_BASE_DIR\java\lib\activemq\activemq\ACTIVEMQ_VERSION ActiveMQ is an Apache open source JMS message broker.
ACTIVEMQ_VERSION	ACTIVEMQ_VERSION sets the version of ActiveMQ installed by Artix.

Table 1: *Artix Java Environment Variables*

Variable	Description
SPRING_CONTAINER_HOME	SPRING_CONTAINER_HOME specifies the directory path to the Artix Spring container. The default location is: ARTIX_JAVA_HOME\containers\ spring_container
IT_WSDLGEN_CONFIG_FILE	IT_WSDLGEN_CONFIG_FILE specifies the configuration used by the Artix WSDLGen code generator. If this variable is not set, you will be unable to run WSDLGen. This variable is required for an Artix Devopment installation. The default location is: IT_ARTIX_BASE_DIR\tools\etc\wsdlgen.cfg Do not modify the default WSDLGen configuration file.
PATH	The Artix bin directories are prepended on the PATH to ensure that the proper libraries, configuration files, and utility programs are used. The default Artix bin directory is: **UNIX** $ARTIX_JAVA_HOME/bin **Windows** %ARTIX_JAVA_HOME%\bin

Customizing your Environment Script

Overview

The `artix_java_env` script sets the Artix Java environment variables using values obtained from the Artix installer. The script checks each one of these settings in sequence, and updates them, where appropriate.

The `artix_java_env` script is designed to suit most needs. However, if you want to customize it for your own purposes, please note the points made in this section.

Before you begin

You can only run the `artix_java_env` script once in any console session. If you run this script a second time, it exits without completing. This prevents your environment from becoming bloated with duplicate information (for example, on your `PATH` and `CLASSPATH`). In addition, if you introduce any errors when customizing the `artix_java_env` script, it also exits without completing.

This feature is controlled by the `ARTIX_JAVA_ENV_SET` variable, which is local to the `artix_java_env` script. `ARTIX_JAVA_ENV_SET` is set to `true` the first time you run the script in a console; this causes the script to exit when run again.

Environment variables

The following applies to the environment variables set by the `artix_java_env` script:

- The `JAVA_HOME` environment variable defaults to the value obtained from the Artix installer. If you do not manually set this variable before running `artix_java_env`, it takes its value from the installer. The default location for the JRE supplied with Artix is `IT_ARTIX_BASE_DIR\jre`.
- The following environment variables are all set with default values relative to `IT_ARTIX_BASE_DIR`:
  - `ANT_HOME`
  - `ACTIVEMQ_HOME`

  If you do not set these variables manually, `artix_java_env` sets them with default values based on `IT_ARTIX_BASE_DIR`. For example, the default for `ANT_HOME` on Windows is `IT_ARTIX_BASE_DIR\tools\ant`.

- The `IT_WSDLGEN_CONFIG_FILE` environment variable is a required only for an Artix Development installation. All other environment variables are required for both Development and Runtime installations.

CHAPTER 2

Artix Java Configuration

This chapter introduces the main concepts and components in the Artix Java runtime configuration. It also describes how you make your configuration available to the runtime.

In this chapter

This chapter includes the following sections:

Artix Java Configuration Files

Overview

The Artix Java runtime adopts an approach of zero configuration or configuration by exception. In other words, configuration is required only if you want to customize the runtime to exhibit non-default behavior.

Artix Java configuration files

The Artix Java runtime supports a number of configuration methods should you want to change the default behavior, enable specific functionality or fine-tune a component's behavior. The supported configuration methods include XML configuration files, WS-Policy and WSDL extensions. XML configuration files are, however, the most versatile way to configure the Artix Java runtime and are the recommended approach to use.

Example 1 shows a simplified example of an Artix Java configuration file.

Example 1: *Artix Java Configuration File*

```
1  <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
       beans http://www.springframework.org/schema/beans/
       spring-beans-2.0.xsd">

2  <!-- your configuration goes here! -->

   </beans>
```

The code shown in Example 1 can be explained as follows:

1. An Artix Java configuration file is really a Spring XML file. You must include an opening Spring `<beans>` element that declares the namespaces and schema files for the child elements that are encapsulated by the `<beans>` element.
2. The contents of your configuration depends on the behavior you want the Artix Java runtime to exhibit. You can, however, use:
   - A simplified beans syntax—that is, the child elements of the Spring `<beans>` element can be any one of a number of custom namespaces. For example, you can use `<jaxws:endpoint xmlns:jaxws="http://cxf.apache.org/jaxws"/>` elements.

- Plain Spring XML—that is, the child elements of the Spring `<beans>` element are Spring `<bean>` elements as defined by the Spring beans 2.0 schema, http://www.springframework.org/schema/beans/spring-beans-2.0.xsd.

Spring framework

Spring is a layered Java/J2EE application framework. Artix Java configuration is based on the Spring core and uses the principles of *Inversion of Control* and *Dependency Injection*.

For more information on the Spring framework, see www.springframework.org. Of particular relevance is chapter 3 of the Spring reference guide*, The IoC container*, which can be found at: http://static.springframework.org/spring/docs/2.0.x/reference/beans.html

For more information on inversion of control and dependency injection, see http://martinfowler.com/articles/injection.html

Artix Java configuration options

For detailed information on the configuration options available for the Artix Java runtime, see Artix Configuration Reference, Java Runtime.

Making Your Configuration File Available

Overview

You can make your configuration file available to the Artix Java runtime in any one of the following ways:

- Name it `cxf.xml` and add it your `CLASSPATH`.
- Use one of the following command-line flags to point to the configuration file. This allows you to save your configuration file anywhere on your system and avoid having to add it to your `CLASSPATH`. It also means you can give your configuration file any name you want:

```
-Dcxf.config.file=<myCfgResource>
```

```
-Dcxf.config.file.url=<myCfgURL>
```

  This is a useful approach for portable JAX-WS applications, for example. It is also the method used in most of the Artix Java samples. For example, in the WS-Addressing sample, located in the *ArtixInstallDir*`/java/samples/ws-addressing` directory, the server start command specifies the `server.xml` configuration file as follows:

```
java -Dcxf.config.file=server.xml
     demo.ws_addressing.server.Server
```

  In this example, the start command is run from the directory in which the `server.xml` file resides.

- Programmatically, by creating a bus and passing the configuration file location as either a URL or string, as follows:

```
(new SpringBusFactory()).createBus(URL myCfgURL)
```

```
(new SpringBusFactory()).createBus(String myCfgResource)
```

CHAPTER 3

Deploying to the Spring Container

Artix provides a Spring container into which you can deploy any Spring-based application, including an Artix Web service endpoint. This document outlines how to deploy and manage an Artix Web service endpoint in the Spring container.

In this chapter

This document discusses the following topics:

Introduction

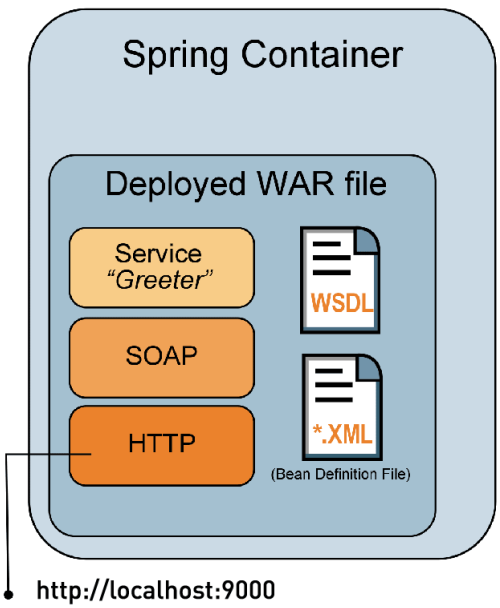
Overview

Artix includes a Spring container that is a customized version of the *Spring framework*. The Spring framework is a general purpose environment for deploying and running Java applications. For more information on the framework, see www.springframework.org. This document explains how to deploy and manage Artix Web service endpoints in the Spring container.

Graphical representation

Figure 1 illustrates how you expose an Artix Web service endpoint from the Spring container.

Figure 1: *Exposing an Artix Web Service Endpoint from the Spring Container*



Essentially, you deploy a WAR file to the Spring container. The WAR file contains all of the files that the Spring container needs to run your application, including the WSDL file that defines your service, the code that you generated from that WSDL file, including the implementation file, any libraries that your application needs and an Artix runtime Spring-based XML configuration file to configure your application.

Sample code

The example code used in this document is taken from the Spring container sample application located in the following directory of your Artix installation:

ArtixInstallDir/java/samples/spring_container/hello_world

For information on how to run this sample, refer to the `README.txt` file in that directory.

Deploying an Artix Endpoint

Deployment steps

The following steps outline, at a high-level, what you must do to successfully configure and deploy an Artix endpoint to the Spring container:

1. Write an Artix Java runtime XML configuration file for your application. See "Configuring your application".
2. Build a WAR file that contains the configuration file, the WSDL file that defines your service, and the code that you generated from that WSDL file, including the implementation file, and any libraries that your application needs. See "Building a WAR file" on page 29.
3. Deploy the WAR file in one of three ways:
   i. Copy the WAR file to the Spring container repository. See "Deploying the WAR file to the Spring repository" on page 29.
   ii. Using the JMX console. See "Managing the Container using the JMX Console" on page 30.
   iii. Using the Web service interface. See "Managing the Container using the Web Service Interface" on page 33.

Configuring your application

You must write an XML configuration file for your application. The Spring container needs this file to instantiate, configure and assemble the beans in your application.

Example 2 shows the configuration file used in the Spring container sample application. It is called `spring.xml`. You can, however, use any name for your configuration file as long as it ends with a `.xml` extension.

Example 2: *Configuration file—spring.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
        Copyright (c) 1993-2006 IONA Technologies PLC.
                        All Rights Reserved.
-->

1  <beans
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Example 2: *Configuration file—spring.xml*

```
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/beans

   http://www.springframework.org/schema/beans/spring-beans.xsd"
    >

2    <jaxws:endpoint id="SoapEndpoint"
        implementor="#SOAPServiceImpl"
        address="http://localhost:9000/SoapContext/SoapPort"
        wsdlLocation="hello_world.wsdl"
        endpointName="e:SoapPort"
        serviceName="s:SOAPService"
        xmlns:e="http://apache.org/hello_world_soap_http"
        xmlns:s="http://apache.org/hello_world_soap_http"/>

3    <bean id="SOAPServiceImpl"
   class="demo.hw.server.GreeterImpl"/>
</beans>
```

The code shown in Example 2 can be explained as follows:

1. The Spring `<beans>` element is required at the beginning of every Artix Java configuration file. It is the only Spring element that you need to be familiar with.
2. Configures a `jaxws:endpoint` that defines a service and its corresponding endpoints. The `jaxws:endpoint` element has the following properties:
   i. `id`—sets the endpoint name or id.
   ii. `implementor`—specifies the implementation object used by the service endpoints. In this case, the configuration file references the `SOAPServiceImpl` bean, which is defined later in the configuration file (see **3** below).
   iii. `address`—specifies the address of the endpoint as defined in the WSDL file that defines service that is being deployed. In this case, `http://localhost:9000/SoapContext/SoapPort`, which is specified in `wsdl:service` element in the `hello_world.wsdl` file (see **iv** below for more detail on `hello_world.wsdl`).

iv. `wsdlLocation`—specifies the WSDL file that contains the service definition. The WSDL file location is relative to the `WEB-INF/wsdl` directory in the WAR file. If you want to look at the `hello_world.wsdl` file used in this example, you can find a copy of it in the *`ArtixInstallDir`*`/java/samples/spring_container/hello_world/wsdl` directory.

v. `endpointName`—specifies the name of port on which the service will run. This value is taken from the WSDL file that defines the service that is being deployed. In this example, the value shown is taken from the `hello_world.wsdl` file. See the `wsdl:port` element in that file.

vi. `serviceName`—specifies the name of service. This value is taken from the WSDL file that defines the service that is being deployed. In this example, the value shown is taken from the `hello_world.wsdl` file. See the `wsdl:service` element in that file.

vii. `xmlns:e`—specifies the namespace of the port. This value is taken from the WSDL file that defines the service that is being deployed. In this example, the value shown is taken from the `hello_world.wsdl` file. See the `targetNamespace` element in that file.

viii. `xmlns:s`—specifies the namespace of the service. This value is taken from the WSDL file that defines the service that is being deployed. In this example, the value shown is taken from the `hello_world.wsdl` file. See the `targetNamespace` property in the `wsdl:definitions` element in that file.

For more information on configuring an Artix `jaxws:endpoint` element, see "Artix Java Configuration" on page 19 and Artix Configuration Reference, Java Runtime.

3. Identifies the class that implements the service.

Building a WAR file

In order to deploy your application to the Spring container you must build a WAR file that has the following structure and contents:

1. `META-INF/spring` should include your configuration file. The configuration file must have a `.xml` extension.
2. `WEB-INF/classes` should include your Web service implementation class and any other classes (including the class hierarchy) generated by the `artix wsdl2java` utility. For information using the `artix wsdl2java` utility, see the Developing Artix Applications with JAX-WS guide.
3. `WEB-INF/wsdl` should include the WSDL file that defines the service that you are deploying.
4. `WEB-INF/lib` should include any JARs required by your application.

Deploying the WAR file to the Spring repository

The simplest way to deploy an Artix endpoint to the Spring container is to:

1. Start the Spring container by running the following command from the *`ArtixInstallDir`*`/java/bin` directory:

   ```
   spring_container start
   ```

2. Copy the WAR file to the Spring container repository. The default location for the repository is:

   `ArtixInstallDir``/java/containers/spring_container/repository`

The Spring container automatically scans the repository for newly deployed applications. The default value at which it scans the repository is every `5000` milliseconds.

Changing the interval at which the Spring container scans its repository

You can change the time interval at which the Spring container scans the repository by changing the `scanInterval` property in the `spring_container.xml` configuration file. See Example 3 on page 34 for more detail.

Changing the default location of the container repository

You can change the Spring container repository location by changing the value of the `containerRepository` property in the `spring_container.xml` configuration file. See Example 3 on page 34 for more detail.

Managing the Container using the JMX Console

Overview

You can use the JMX console to deploy and manage applications in the Spring container. Specifically you can deploy applications as well as stop, start, remove, and list applications that are running in the container. You can also get information on the application's state. The name of the deployed WAR file is the name given to the application.

Using the JMX console

To use the JMX console to manage applications deployed to the Spring container do the following:

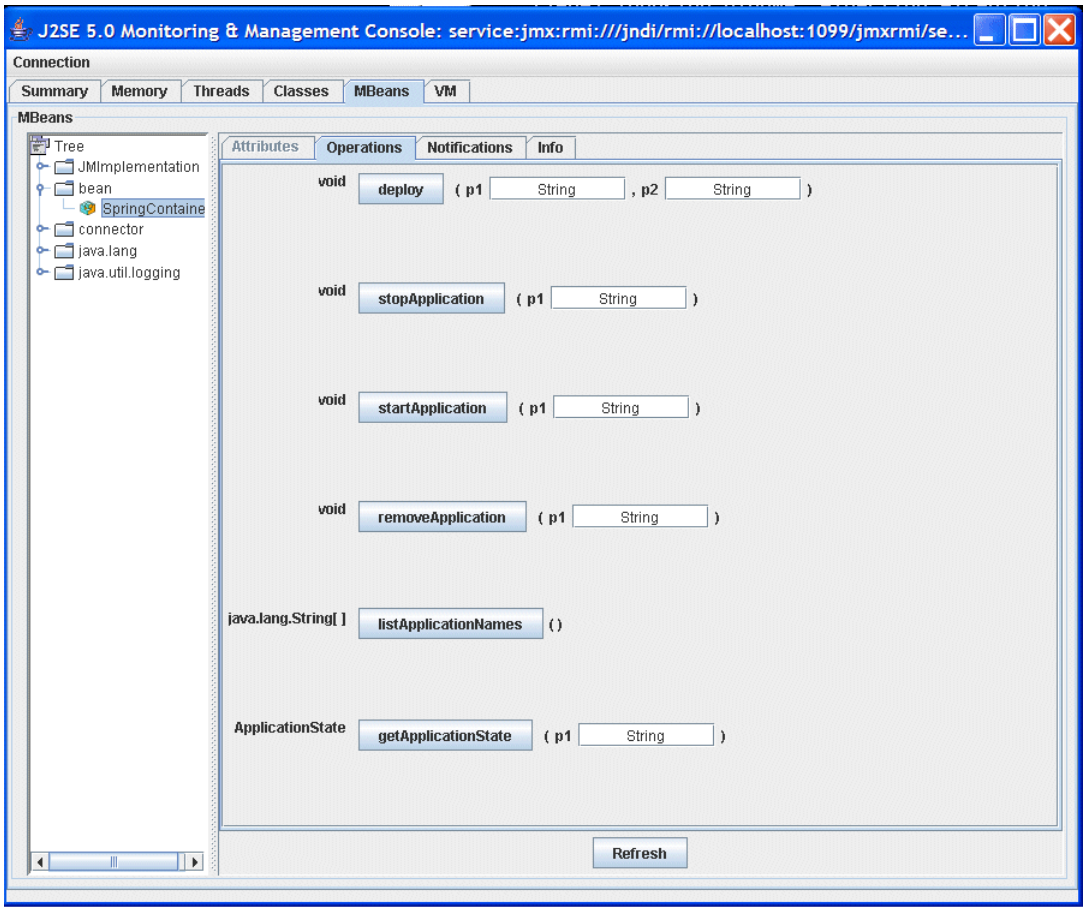
1. Start the JMX console by running the following command from the *`ArtixInstallDir`*`/java/bin` directory:

   Windows: `jmx_console_start.bat`

   UNIX: `jmx_console_start.sh`

2. Select the **MBeans** tag and expand the **bean** node to view the `SpringContainer` MBean (see Figure 2 on page 31). The `SpringContainer` MBean is deployed as part of the Spring container. It gives you access to the management interface for the Spring Container and can be used to deploy, stop, start, remove and list applications, and get information on an application's state.

Figure 2: *JMX Console—SpringContainer MBean*



The operations and their parameters are described in Table 2:

Table 2: *JMX Console—SpringContainer MBean Operations*

Operation	Description	Parameters
`deploy`	Deploys an application to the container repository. The `deploy` method copies a WAR file from a given URL or file location into the container repository.	`location`—a URL or file location that points to the application to be deployed. `warFileName`—the name of the WAR file as you want it to appear in the container repository.
`stopApplication`	Stops the specified application. It does not remove the application from the container repository.	`name`—specifies the name of the application that you want to stop. The application name is the same as the WAR file name.
`startApplication`	Starts an application that has previously been deployed and subsequently stopped.	`name`—specifies the name of the application that you want to start. The application name is the same as the WAR file name.
`removeApplication`	Stops and removes an application. It completely removes an application from the container repository.	`name`—specifies the name of the application that you want to stop and remove. The application name is the same as the WAR file name.
`listApplicationNames`	Lists all of the applications that have been deployed. The applications can be in one of three states: start, stop, or failed. An application's name is the same as its WAR file name.	None.
`getApplicationState`	Reports the state of an application; that is, whether it is running or not.	`name`—specifies the name of the application whose state you want to know. The application name is the same as the WAR file name.

Managing the Container using the Web Service Interface

Overview

You can use the Web service interface to deploy and manage applications in the Spring container. The Web service interface is specified in the `container.wsdl` file, which is located in the *`ArtixInstallDir`*`/java/containers/spring_container/etc/wsdl` directory of your Artix installation.

Client tool

At present Artix does not include a client tool for the Web service interface. You could, however, write one if you are familiar with Web service development. Please refer to the `container.wsdl` file and the Artix Java runtime Javadoc for more detail.

Changing the port on which the Web service interface listens

To change the port on which the Web service interface listens, you must change the port number of the address property in the `spring_container.xml` file; that is, `<jaxws:endpoint id="ContainerService" implementor="#ContainerServiceImpl" address="http://localhost:2222/AdminContext/AdminPort" ...>`. You do not need to change the `container.wsdl` file.

For more information on the `spring_container.xml` file, see "Spring Container Definition File" on page 34.

Adding a port

If you want to add a port, such as a JMS port or an HTTPS port, add the port details to the `container.wsdl` file.

Spring Container Definition File

Overview

The Spring container is configured in the `spring_container.xml` file located in the following directory of your Artix installation:

ArtixInstallDir`/java/containers/spring_container/etc`

spring_container.xml

The contents of the Spring container configuration file are shown in Example 3:

Example 3: *spring_container.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Copyright (c) 1993-2006 IONA Technologies PLC.
          All Rights Reserved.
-->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:container="http://schemas.iona.com/soa/
       container-config"
       xsi:schemaLocation="http://www.springframework.org/
                           schema/beans
                            http://www.springframework.org/schema/
                           beans/spring-beans-2.0.xsd
                          http://cxf.apache.org/jaxws
                           http://cxf.apache.org/schemas/jaxws.xsd
                            http://schemas.iona.com/soa/
                           container-config
                            http://schemas.iona.com/soa/
                           container-config.xsd">

   <!-- Bean definition for Container -->

1  <container:container id="container"
   containerRepository="C:\Artix_5.0\java/containers/spring_cont
   ainer/repository" scanInterval="5000"/>

   <!-- Web Service Container Management -->

2  <jaxws:endpoint id="ContainerService"
```

Example 3: *spring_container.xml*

```
   implementor="#ContainerServiceImpl"
   address="http://localhost:2222/AdminContext/AdminPort"
   wsdlLocation="/wsdl/container.wsdl"
   endpointName="e:ContainerServicePort"
   serviceName="s:ContainerService"
   xmlns:e="http://cxf.iona.com/container/admin"
   xmlns:s="http://cxf.iona.com/container/admin"/>

<bean id="ContainerServiceImpl"
   class="com.iona.cxf.container.admin.ContainerAdminServiceImpl
   ">
        <property name="container">
            <ref bean="container" />
        </property>
</bean>

    <!--  JMX Container Management  -->

3  <bean id="mbeanServer"
   class="org.springframework.jmx.support.MBeanServerFactoryBean
   ">
       <property name="locateExistingServerIfPossible"
        value="true" />
   </bean>

   <bean id="exporter"
   class="org.springframework.jmx.export.MBeanExporter">
       <property name="beans">
           <map>
               <entry key="bean:name=SpringContainer"
               value-ref="container"/>
               <entry key="connector:name=rmi"
               value-ref="serverConnector"/>
           </map>
       </property>

       <property name="server" ref="mbeanServer"/>
       <property name="assembler" ref="assembler" />
   </bean>

   <bean id="assembler"
   class="org.springframework.jmx.export.assembler.InterfaceBase
   dMBeanInfoAssembler">
       <property name="interfaceMappings">
           <props>
```

Example 3: *spring_container.xml*

```
                <prop key="bean:name=SpringContainer">
                com.iona.cxf.container.managed.JMXContainer</prop>
            </props>
        </property>
    </bean>

    <bean id="serverConnector"
    class="org.springframework.jmx.support.ConnectorServerFactory
    Bean"  depends-on="registry" >
        <property name="serviceUrl"
         value="service:jmx:rmi:///jndi/rmi://localhost:1099/
         jmxrmi/server"/>
    </bean>

    <bean id="registry"
    class="org.springframework.remoting.rmi.RmiRegistryFactoryBea
    n">
        <property name="port" value="1099"/>
    </bean>

</beans>
```

The code shown in Example 3 can be explained as follows:

1. Defines a bean that encapsulates the logic for the Spring container. This bean handles the logic for deploying user applications that are copied to the specified container repository location. The default container repository location is: *ArtixInstallDir*`/java/containers/spring_container/repository`. You can change the repository location by changing the value of the `containerRepository` property.

   The `scanInterval` property sets the time interval at which the repository is scanned. It is set in milliseconds. The default value is set to 5000 milliseconds. Removing this attribute disables scanning.
2. Defines an application that creates a Web service interface that you can use to manage the Spring container.

   The `ContainerServiceImpl` bean contains the server implementation code and the container administration logic.

To change the port on which the Web service interface listens, change the `address` property; that is, `address="http://localhost:2222/AdminContext/AdminPort"`.

3. Defines Spring beans that allow you to use a JMX console to manage the Spring container.

   For more information, please refer to the JMX chapter of the Spring 2.0.x reference document available at:

   http://static.springframework.org/spring/docs/2.0.x/reference/jmx.html

Running Multiple Containers on Same Host

Overview

If you want to run more than one Spring container on the same host you must complete the steps outlined below:

1. Make a copy of the `spring_container.xml` file, which is located in the *`ArtixInstallDir`*`/java/containers/spring_container/etc` directory.
2. Make the following changes to your new copy, *`my_spring_container`*`.xml`:
   i. Container repository location—change the following line:

```
<container:container id="container"
   containerRepository="c:\artix50\java/containers/spring_contai
   ner/repository" scanInterval="5000"/>
```

   to point to a new container repository. For example:

```
<container:container id="container"
   containerRepository="c\:artix50\java/containers/spring_contai
   ner/repository2" scanInterval="5000"/>
```

   ii. Change the port on which the Web service interface listens to something other than `2222` by changing the `address` property shown below:

```
<jaxws:endpoint id="ContainerService"
   implementor="#ContainerServiceImpl" address="
   http://localhost:2222/AdminContext/AdminPort"
```

   iii. Change the JMX port to something other than `1099` in the following line:

```
<bean id="serverConnector"
   class="org.springframework.jmx.support.ConnectorServerFactory
   Bean"  depends-on="registry" >
    <property name="serviceUrl"
   value="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/se
   rver"/>
</bean>
```

iv. Change the RMI registry port to something other than `1099` in the following line:

```
<bean id="registry"
   class="org.springframework.remoting.rmi.RmiRegistryFactoryBea
   n">
    <property name="port" value="1099"/>
</bean>
```

3. Make a copy of the JMX console launch script, `jmx_console_start.bat`, which is located in the *`ArtixInstallDir`*`/java/bin` directory.
4. Change the following line in your copy of the JMX console launch script to point to the JMX port that you specified in step 2 (iii) above:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server
```

5. Start the new container by passing the URL or file location of its configuration file, `my_spring_container.xml`, to the `start_container.bat` script. The syntax of the `start_container.bat` file is:

```
spring_container -config <spring-config-url> -h -verbose
   <start|stop>
```

For example, where the `my_spring_container.xml` file has been saved to the *`ArtixInstallDir`*`/java/containers/spring_container/etc` directory, run the following command:

```
ArtixInstallDir/java/bin/spring_container.bat -config
   ..\containers\spring_container\etc\my_spring_container.xml
   start
```

6. To view the new container using the JMX console, run the JMX console launch script that you created in steps 3 and 4 above.
7. To stop the new container, use `Ctrl-C`.

CHAPTER 4

Deploying to a Servlet Container

You can deploy and run an Artix Web service endpoint in any servlet container. This document explains how.

In this chapter

This document discusses the following topics:

Introduction

Overview

You can deploy and run an Artix Web service endpoint from any servlet container. Artix provides a standard servlet, the CXF servlet, which acts as an adapter for the Web service endpoints.

Sample application

This document uses, as an example, the Artix servlet container sample application that is included in the following directory of your Artix installation:

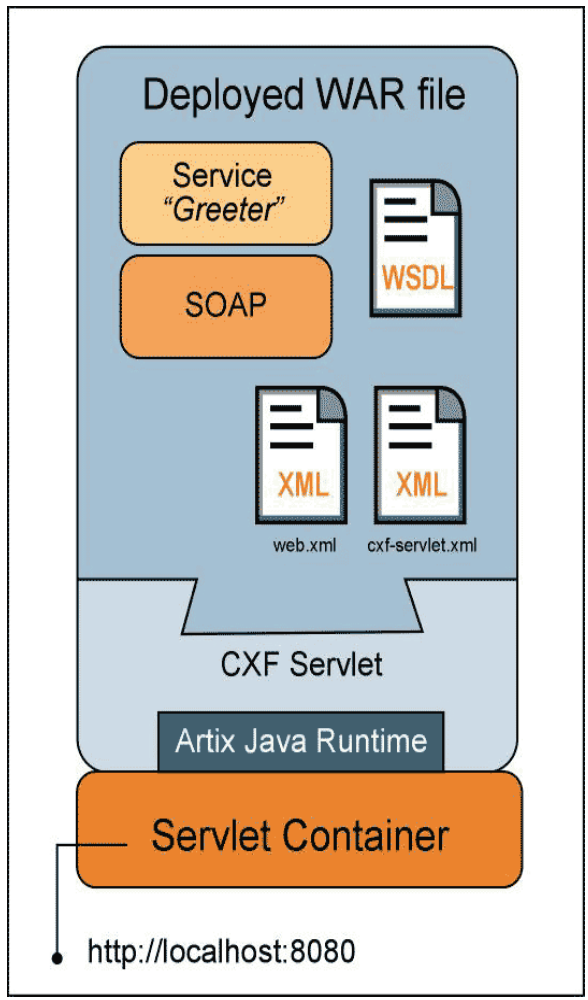
ArtixInstallDir`/java/samples/hello_world`

For information on how to run this sample application, see the `README.txt` in that directory.

Graphical overview

Figure 3 on page 43 shows the main components of the servlet container sample application and illustrates how you can expose an Artix Web service endpoint from a servlet container:

Figure 3: *Exposing an Artix Web Service Endpoint from a Servlet Container*



Servlet container	The servlet container shown in Figure 3 on page 43 can be any servlet container. All hosted services are accessed through the same IP port; for example, if you use Tomcat, the default IP port is `8080`.
Deployed WAR file	Services are deployed to the servlet container in a Web Archive (WAR) file, as shown in Figure 3 on page 43. The deployed WAR file contains the compiled code for the service being deployed, a copy of the WSDL file that defines the service, the WSDL stub code, and configuration files.
CXF servlet	The CXF servlet shown in Figure 3 on page 43 is a standard servlet provided by Artix. It acts as an adapter for Web service endpoints and is part of the Artix Java runtime. It is implemented by the `org.apache.cxf.transport.servlet.CXFServlet` class.
cxf-servlet.xml file	The `cxf-servlet.xml` file configures the endpoints that plug into the CXF servlet.
web.xml file	The `web.xml` file is a standard deployment descriptor file that tells the servlet container to load the `org.apache.cxf.transport.servlet.CXFServlet` class.

Configuring Servlet Container to Run an Artix Application

Overview

Before you can deploy an Artix Web service endpoint to your servlet container you must configure the servlet container so that it can run Artix applications. How you do this depends on the servlet container that you are using. This section highlights the key configuration steps that you must complete and uses Tomcat as an example servlet container.

Making certain Artix JARs available to your application

You need to make all of the JAR files in the *ArtixInstallDir*/java/lib directory available to your application. The only exception is the *jbi*.jar files.

If, for example, you are using Tomcat 5.x or lower, copy the JAR files from your *ArtixInstallDir*/java/lib directory to your CATALINA_HOME/shared/lib directory.

If, however, you are using Tomcat 6, copy the JAR files from your *ArtixInstallDir*/java/lib directory to your CATALINA_HOME/lib directory.

Note: You must restart Tomcat after you copy the Artix JAR files to the CATALINA_HOME/shared/lib or CATALINA_HOME/lib directory. Tomcat does not dynamically pick up the JAR files.

Making the Artix licenses.txt file available to Artix Java runtime

You must make sure that the Artix licenses.txt file is available to the servlet container. For example, if you are using Tomcat, copy the Artix licenses.txt file from the *ArtixInstallDir*/etc directory, where it is stored by default, to the CATALINA_HOME/shared/classes directory.

Using ant to automate servlet container configuration

The Artix Java samples directory, *ArtixInstallDir*/java/samples, includes a common_build.xml file that contains an ant target that copies the Artix JAR files to CATALINA_HOME/shared/lib and the Artix licenses.txt file to the CATALINA_HOME/shared/classes directory. The relevant ant target is shown in Example 4:

Example 4: *common.xml—Ant Target that Configures Tomcat*

```
<target name="prepare.tomcat"
   unless="cxf.jars.present.in.tomcat">
      <copy todir="${env.CATALINA_HOME}/shared/lib">
          <fileset dir="${cxf.home}/lib">
              <include name="*.jar"/>
              <exclude name="*jbi*.jar" />
          </fileset>
      </copy>
      <copy file="${cxf.home}/../etc/licenses.txt"
   todir="${env.CATALINA_HOME}/shared/classes"/>
</target>
```

This common_build.xml file is included in the build.xml file that is used to build and run the Artix Java servlet container sample application, which is contained in the *ArtixInstallDir*/java/samples/hello_world directory.

Deploying an Artix Endpoint

Deployment steps

The following outlines, at a high-level, what you must do to successfully deploy an Artix Web service endpoint to a servlet container:

1. Build a WAR file that contains the your application, the WSDL file that defines your service, a `web.xml` deployment descriptor file that tells the servlet container to load the CXF servlet class, and a `cxf-servlet.xml` deployment descriptor file that configures the endpoints that plug into the CXF servlet.
2. Deploy the WAR file to your servlet container.

Building a WAR file

In order to deploy your application to a servlet container you must build a WAR file that has the following directories and files:

1. `WEB-INF` should include a:
   i. `cxf-servlet.xml` file—which configures the endpoints that plug into the CXF servlet. When the CXF servlet starts up, it reads the `jaxws:endpoint` elements from this file and initializes a service endpoint for each one. See Example 5 on page 48 for more information.
   ii. `web.xml` file—which instructs the servlet container to load the `org.apache.cxf.transport.servlet.CXFServlet` class. A reference version of this file is contained in your *`ArtixInstallDir`*`/java/etc` directory. You can use this reference copy and do not need to make any changes to it.
2. `WEB-INF/classes` should include your Web service implementation class and any other classes (including the class hierarchy) generated by the `artix wsdl2java` utility.
3. `WEB-INF/wsdl` should include the WSDL file that defines the service that you are deploying.
4. `WEB-INF/lib` should include any JARs required by your application.

Example cxf-servlet.xml file

The `cxf-servlet.xml` file configures the endpoints that plug into the CXF servlet. When the CXF servlet starts up it reads the list of `jaxws:endpoint` elements in this file and initializes a service endpoint for each one.

Example 5 shows the `cxf-servlet.xml` file used in the Artix servlet container sample application. It contains one `jaxws:endpoint` element that configures the Greeter service endpoint.

Example 5: *cxf-servlet.xml file*

```
<?xml version="1.0" encoding="UTF-8"?>
1 <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:soap="http://cxf.apache.org/bindings/soap"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.
   xsd
http://cxf.apache.org/bindings/soap
   http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws
   http://cxf.apache.org/schemas/jaxws.xsd">

2    <jaxws:endpoint
        id="hello_world_xml_bare"
        implementor="demo.hw.server.GreeterImpl"
        wsdlLocation="WEB-INF/wsdl/hello_world.wsdl"
        address="/hello_world">
             <jaxws:features>
                  <bean class=
                   "org.apache.cxf.feature.LoggingFeature"/>
             </jaxws:features>
    </jaxws:endpoint>
</beans>
```

The code shown in Example 5 on page 48 can be explained as follows:

1. The Spring `<beans>` element is required at the beginning of every Artix Java configuration file. It is the only Spring element that you need to be familiar with.
2. Configures a `jaxws:endpoint` that defines a service and its corresponding endpoints. The `jaxws:endpoint` element has the following properties:
   i. `id`—sets the endpoint name or id.
   ii. `implementor`—specifies the implementation object used by the service endpoints. In this case, the configuration file references the `demo.hw.server.GreeterImpl` bean, which is included in the application WAR file.
   iii. `wsdlLocation`—specifies the WSDL file that contains the service definition. The WSDL file location is relative to the `WEB-INF/wsdl` directory in the WAR file. If you want to look at the `hello_world.wsdl` file used in this example, you can find a copy of it in the *`ArtixInstallDir`*`/java/samples/spring_container/hello_world/wsdl` directory.
   iv. `address`—specifies the address of the endpoint as defined in the WSDL file that defines service that is being deployed. In this case, `http://localhost:9000/SoapContext/SoapPort`, which is specified in `wsdl:service` element in the `hello_world.wsdl` file.
   v. The `jaxws:features` element defines features that can be added to your endpoint. In this example the logging feature is added to the endpoint deployed.

      For more information on the `jaxws:features` element, see "Artix Java Configuration" on page 19.

      For more information on logging, see "Artix Logging" on page 51

Reference web.xml file

You must include a `web.xml` deployment descriptor file that tells the servlet container to load the CXF servlet. Example 6 shows the `web.xml` file that is used in the Artix servlet container sample application. You do not need to change this file. A reference copy is located in *`ArtixInstallDir`*`/java/etc` directory.

Example 6: *web.xml deployment descriptor file*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
   Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
   <display-name>cxf</display-name>
   <description>cxf</description>
   <servlet>
        <servlet-name>cxf</servlet-name>
        <display-name>cxf</display-name>
        <description>Apache CXF Endpoint</description>
        <servlet-class>org.apache.cxf.transport.servlet.
        CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>cxf</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

Deploying WAR file to the servlet container

How you deploy your WAR file depends on the servlet container that you are using. For example, to deploy your WAR file to Tomcat, copy it to the Tomcat `CATALINA_HOME/server/webapp` directory.

If you are using a different servlet container, please refer to the deployment documentation for that container.

CHAPTER 5

Artix Logging

This chapter describes how to configure logging in the Artix Java runtime.

In this chapter

This chapter includes the following sections:

Overview of Artix Java Logging

Overview

The Artix Java runtime uses the Java logging utility, `java.util.logging`. Logging is configured in a logging configuration file that is written using the standard `java.util.Properties` format. To run logging on an application, you can specify logging programmatically or by defining a property at the command that points to the logging configuration file when you start the application.

Default logging.properties file

The Artix Java runtime comes with a default `logging.properties` file. The `logging.properties` file is located in your *`ArtixInstallDir`*`/java/etc` directory. This file configures the output destination for the log messages and the message level that is published. The default `logging.properties` file configures the Artix Java loggers to print log messages of level `WARNING` to the console. You can use this file without changing any of the configuration settings or you can change the configuration settings to best suit your application.

Logging feature

The Artix Java runtime includes a logging feature that you can plugin to your client or service to enable logging. The following configuration enables the logging feature:

```
<jaxws:endpoint...>
    <jaxws:features>
        <bean class="org.apache.cxf.feature.LoggingFeature"/>
    </jaxws:features>
</jaxws:endpoint>
```

For more information, see "Logging Message Content" on page 65.

Where to begin?

If you want to run a simple example of logging without the need to understand everything about logging, follow the instructions outlined in a "Simple Example of Using Logging" on page 54.

For more information on how logging works in the Artix Java runtime, read the entire chapter.

More information on java.util.logging

The `java.util.logging` utility is one of the most widely used Java logging frameworks. There is a lot of user information available online that describes how to use and extend this framework. As a starting point, however, the following document gives a good overview of `java.util.logging`:

http://java.sun.com/j2se/1.5.0/docs/guide/logging/overview.html

For details on the `java.util.logging` API, see http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/package-summary.html

Simple Example of Using Logging

Overview

This section walks you through a simple example of using logging in the Artix Java runtime.

Changing the log levels and output destination in hello world sample

To change the log level and output destination of the log messages in the hello world sample application, complete the following steps:

1. Run the hello world sample server as described in the *Running the demo using java* section of the `README.txt` file in the *`ArtixInstallDir`*`/java/samples/hello_world` directory. Note that the server start command specifies the default `logging.properties` file, as follows:

   Windows:

```
start java -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\
etc\logging.properties demo.hw.server.Server
```

   UNIX:

```
java -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/
   logging.properties demo.hw.server.Server &
```

   The default `logging.properties` file is located in the *`ArtixInstallDir`*`/java/etc` directory. It configures the Artix Java loggers to print `WARNING` level log messages to the console. As a result, you will see very little printed to the console.
2. Stop the hello world server as described in the `README.txt` file.
3. Make a copy of the default `logging.properties` file, name it `mylogging.properties` file, and save it in the same directory as the default `logging.properties` file; that is, *`ArtixInstallDir`*`/java/etc`

4. Change the global logging level and the console logging levels in your `mylogging.properties` file to `INFO` by editing the following lines of configuration:

```
.level= INFO

java.util.logging.ConsoleHandler.level = INFO
```

5. Restart the hello world server using the following command:

   Windows:

```
start java
   -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\etc\myloggi
   ng.properties demo.hw.server.Server
```

   UNIX:

```
java
   -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/myloggin
   g.properties demo.hw.server.Server &
```

   Because you have configured the global logging and the console logger to log messages of level `INFO`, you will see a lot more log messages printed to the console.

Default logging.properties File

Overview

The default logging configuration file, `logging.properties`, is located in the *`ArtixInstallDir`*`/java/etc` directory. It configures the Artix Java loggers to print `WARNING` level messages to the console. If this level of logging suits your application, you do not have to make any changes to the file before using it. You can, however, change the level of detail in the log messages and whether, for example, the log messages are sent to the console, to a file or to both. In addition, you can specify logging at the level of individual packages.

In this subsection

This section discusses the configuration properties that appear in the default `logging.properties` file. There are, however, many other `java.util.logging` configuration properties that you can set. For more information on the `java.util.logging` API, see the `java.util.logging` javadoc at: http://java.sun.com/j2se/1.5/docs/api/java/util/logging/package-summary.html

Configuring logging output

The Java logging utility, `java.util.logging`, uses handler classes to output log messages. Table 3 shows the handlers that are configured in the default default `logging.properties` file:

Table 3: *Java.util.logging Handler Classes*

Handler Class	Outputs to
`ConsoleHandler`	Outputs log messages to the console.
`FileHandler`	Outputs log messages to a file.

The handler classes must be on the system classpath in order to be installed by the Java VM when it starts. This is done when you set the Artix Java environment. For details on how to set the Artix Java environment, see "Setting your Artix Java Environment" on page 12

Configuring the Console Handler

The following line of code configures the console handler:

```
handlers= java.util.logging.ConsoleHandler
```

Configuring the File Handler

The following line of code configures the file handler:

```
handlers= java.util.logging.FileHandler
```

The file handler also supports the configuration properties shown in Example 7:

Example 7: *File Handler Configuration Properties*

```
1 java.util.logging.FileHandler.pattern = %h/java%u.log
2 java.util.logging.FileHandler.limit = 50000
3 java.util.logging.FileHandler.count = 1
4 java.util.logging.FileHandler.formatter =
     java.util.logging.XMLFormatter
```

The configuration properties shown in Example 7 can be explained as follows:

1. Specifies the location and pattern of the output file. The default setting is your home directory.
2. Specifies, in bytes, the maximum amount that the logger writes to any one file. The default setting is `50000`. If you set to zero, there is no limit on the amount that the logger writes to any one file.
3. Specifies how many output files to cycle through. The default setting is `1`.
4. Specifies the `java.util.logging` formatter class that the file handler class uses to format the log messages. The default setting is the `java.util.logging.XMLFormatter`.

Configuring Both the Console Handler and the File Handler

In addition, you can set the logging utility to output log messages to both the console and to a file by specifying the console handler and the file handler, separated by a comma, as follows:

```
handlers= java.util.logging.FileHandler,
   java.util.logging.ConsoleHandler
```

The console handler also supports the configuration properties shown in Example 8:

Example 8: *Console Handler*

```
1 java.util.logging.ConsoleHandler.level = WARNING
2 java.util.logging.ConsoleHandler.formatter =
     java.util.logging.SimpleFormatter
```

The configuration properties shown in Example 8 can be explained as follows:

1. The console handler supports a separate log level configuration property. This allows you to limit the log messages printed to the console while the global logging setting can be different (see Configuring global logging levels). The default setting is `WARNING`.
2. Specifies the `java.util.logging` formatter class that the console handler class uses to format the log messages. The default setting is the `java.util.logging.SimpleFormatter`.

Configuring global logging levels

The `java.util.logging` framework supports the following levels of logging, from least verbose to most verbose:

- `SEVERE`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`

To configure the types of event that are logged across all loggers, configure the global logging level as follows:

```
.level= WARNING
```

The global logging level can be overridden by setting a package-specific log level. For more information, see Configuring logging at an individual package level.

Configuring logging at an individual package level

The `java.util.logging` framework supports configuring logging at the level of an individual package. For example, the following line of code configures logging at a `SEVERE` level on classes in the `com.xyz.foo` package:

```
com.xyz.foo.level = SEVERE
```

Enabling Logging at the Command Line

Overview

You can run the logging utility on an application by defining a `java.util.logging.config.file` property when you start the application. You can specify the default `logging.properties` file or a `logging.properties` file that is unique to that application.

Specifying the log configuration file on application start-up

To specify logging on application start-up run the following command:

```
java -Djava.util.logging.config.file=myfile
```

For example, the following commands start the server in the hello world sample application:

Windows:

```
start java
   -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/logging.
   properties demo.hw.server.Server
```

UNIX:

```
java
   -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/logging.
   properties demo.hw.server.Server &
```

Note that the start command specifies the default `logging.properties` file.

For more information on running the hello world sample application and to see Artix Java logging working, see "Simple Example of Using Logging" on page 54.

Logging for Subsystems and Services

Overview

You can use the `com.xyz.foo.level` configuration property described in "Configuring logging at an individual package level" on page 59 to set fine-grained logging for specified Artix Java logging subsystems.

Artix Java logging subsystems

Table 4 shows a list of available Artix Java logging subsystems:

Table 4: *Artix Java Logging Subsystems*

Subsystem	Description
`com.iona.cxf.container`	Artix Java container.
`com.iona.cxf.ha.failover`	Artix Java high availability services back-end.
`com.iona.cxf.locator`	Artix Java locator client and endpoint, which provide support for communicating with the Artix locator.
`com.iona.cxf.management.amberpoint`	Artix Java AmperPoint integration.
`com.iona.cxf.management.bmc`	Artix Java BMC Patrol integration.
`com.iona.cxf.peer_manager`	Peer manager component of high availability implementation.
`com.iona.cxf.security`	Artix Java security service. Extends WSS4j security for secure web services and clients.
`com.iona.cxf.transport.ftp`	Artix Java FTP transport.
`com.iona.cxf.wsdlpublish`	Artix Java WSDL publishing service.
`org.apache.cxf.aegis`	Artix Java Aegis binding.
`org.apache.cxf.binding.coloc`	Artix Java colocated binding.
`org.apache.cxf.binding.http`	Artix Java HTTP binding.
`org.apache.cxf.binding.jbi`	Artix Java JBI binding. For use with JBI containers.

Table 4: *Artix Java Logging Subsystems*

Subsystem	Description
`org.apache.cxf.binding.object`	Artix Java Java Object binding.
`org.apache.cxf.binding.soap`	Artix Java SOAP binding.
`org.apache.cxf.binding.xml`	Artix Java XML binding.
`org.apache.cxf.bus`	Artix Java bus.
`org.apache.cxf.configuration`	Artix Java configuration framework.
`org.apache.cxf.endpoint`	Artix Java server and client endpoints.
`org.apache.cxf.interceptor`	Artix Java interceptors.
`org.apache.cxf.jaxws`	Front-end for JAX-WS style message exchange, JAX-WS handler processing, and interceptors relating to JAX-WS and configuration.
`org.apache.cxf.jbi`	JBI container integration classes.
`org.apache.cxf.jca`	JCA container integration classes.
`org.apache.cxf.js`	Artix Java JavaScript front-end.
`org.apache.cxf.transport.http`	Artix Java HTTP transport.
`org.apache.cxf.transport.https`	Artix Java secure version of HTTP transport, using HTTPS.
`org.apache.cxf.transport.jbi`	Artix Java JBI transport. For integration with JBI container.
`org.apache.cxf.transport.jms`	Artix Java JMS transport.
`org.apache.cxf.transport.local`	Artix Java transport implementation using local file system.
`org.apache.cxf.transport.servlet`	Artix Java HTTP transport and servlet implementation for loading JAX-WS endpoints into a servlet container.
`org.apache.cxf.ws.addressing`	Artix Java WS-Addressing implementation.

Table 4: *Artix Java Logging Subsystems*

Subsystem	Description
`org.apache.cxf.ws.policy`	Artix Java WS-Policy specification implementation. Provides the framework for building and applying WS-Policy policy assertions.
`org.apache.cxf.ws.rm`	Artix Java WS-ReliableMessaging (WS-RM) implementation.
`org.apache.cxf.ws.security.wss4j`	Artix Java WSS4J security implementation.
`org.apache.yoko.bindings.corba`	Artix Java CORBA binding.
`org.apache.yoko.bindings.corba.interceptors`	Artix Java CORBA binding interceptors. Used for intercepting and working with the raw messages.
`org.apache.yoko.bindings.corba.runtime`	Implementation of the CORBA binding runtime. Reading and writing the content of the message.
`org.apache.yoko.tools.common.idltypes`	Artix Java CORBA implementation of the CORBA IDL types.
`org.apache.yoko.tools.processors.wsdl`	Artix Java CORBA binding creation of a WSDL file.

Examples of configuring Artix Java subsystems

Examples of configuring logging for specific Artix Java subsystems can be seen in some of the Artix Java samples. Two such examples are:

- Security
- WS-Addressing

Security

The authentication sample application is contained in the *ArtixInstallDir*`/java/samples/security/authentication` directory. Logging is configured in the `logging.properties` file located in the `etc` directory. The relevant line of configuration is:

```
com.iona.cxf.security.level=INFO
```

For information on running this sample, see the `README.txt` file located in the *ArtixInstallDir*`/java/samples/security/authentication` directory.

WS-Addressing

The WS-Addressing sample is contained in the *ArtixInstallDir*/java/samples/ws_addressing directory. Logging is configured in the logging.properties file located in that directory. The relevant lines of configuration are as follows:

```
java.util.logging.ConsoleHandler.formatter =
    demos.ws_addressing.common.ConciseFormatter
...
org.apache.cxf.ws.addressing.soap.MAPCodec.level = INFO
```

This configuration enables the snooping of log messages relating to WS-Addressing headers and displays them to the console in a concise form.

For information on running this sample, see the README.txt file located in the *ArtixInstallDir*/java/samples/ws_addressing directory.

Logging Message Content

Overview

You can log the content of the messages that are sent between a service and a consumer. For example, you might want to log the contents of SOAP messages that are being sent between a service and a consumer.

Configuring message content logging

To log the messages that are sent between a service and a consumer, complete the following steps:

1. Add the logging feature to your service configuration
2. Configure logging to log INFO level messages

Add the logging feature to your service configuration

Add the logging feature your service configuration as shown in Example 9:

Example 9: *Adding Logging Feature to Your Service Configuration*

```
<jaxws:endpoint ...>

   <jaxws:features>
       <bean class="org.apache.cxf.feature.LoggingFeature"/>
   </jaxws:features>

</jaxws:endpoint>
```

The code shown in Example 9 enables the logging of SOAP messages.

Configure logging to log INFO level messages

Ensure that the `logging.properties` file associated with your service is configured to log `INFO` level messages, as follows:

```
.level= INFO

java.util.logging.ConsoleHandler.level = INFO
```

Logging SOAP message

To see the logging of SOAP messages, for example, modify the hello world sample application, located in the *ArtixInstallDir*/java/samples/hello_world directory, as follows:

1. Add the <jaxws:features> element shown below to the cxf.xml configuration file located in the hello world sample directory:

```
<jaxws:endpoint
   name="{http://apache.org/hello_world_soap_http}SoapPort"
                createdFromAPI="true">
      <jaxws:properties>
          <entry key="schema-validation-enabled" value="true" />
      </jaxws:properties>
      <jaxws:features>
           <bean class="org.apache.cxf.feature.LoggingFeature"/>
      </jaxws:features>
</jaxws:endpoint>
```

2. The hello world sample uses the default logging.properties file, which is located in *ArtixInstallDir*/java/etc directory. Make a copy of this file and call it mylogging.properties file.
3. In the mylogging.properties file, change the logging levels to INFO by editing the .level and java.util.logging.ConsoleHandler.level configuration properties as follows:

```
.level= INFO

java.util.logging.ConsoleHandler.level = INFO
```

4. Start the hello world server using the new configuration settings in the cxf.xml file and the mylogging.properties file as follows:

   Windows:

```
start java
   -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\etc\myloggi
   ng.properties demo.hw.server.Server
```

   UNIX:

```
java
   -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/myloggin
   g.properties demo.hw.server.Server &
```

5. Start the hello world client using the following command:

   Windows:

```
java
   -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\etc\myloggi
   ng.properties demo.hw.client.Client .\wsdl\hello_world.wsdl
```

   UNIX:

```
java
   -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/myloggin
   g.properties demo.hw.client.Client ./wsdl/hello_world.wsdl
```

   The SOAP messages are logged to the console.

CHAPTER 6

Enabling Reliable Messaging

Artix supports Web Services Reliable Messaging (WS-Reliable Messaging). This chapter explains how to enable and configure WS-RM in an Artix Java runtime environment.

In this chapter

This chapter discusses the following topics:

Introduction to WS-RM

Overview

Web Services Reliable Messaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

For example, WS-RM can be used to ensure that the correct messages have been delivered across a network exactly once, and in the correct order. Web Services Reliable Messaging is also known as WS-ReliableMessaging.

How WS-RM works

WS-RM ensures the reliable delivery of messages between a source and destination endpoint. The source is the initial sender of the message and the destination is the ultimate receiver, as shown in Figure 4.

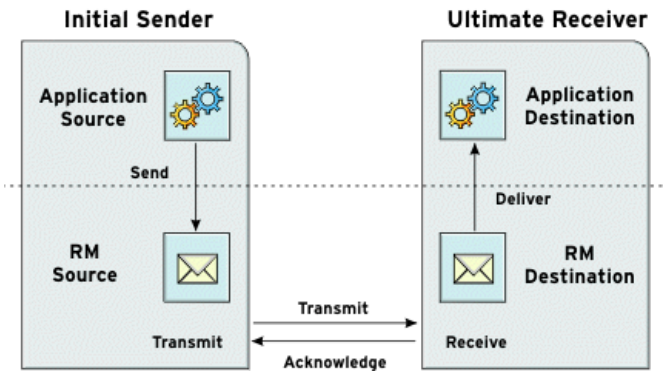


Figure 4: *Web Services Reliable Messaging*

The flow of WS-RM messages can be described as follows:

1. The RM source sends a `CreateSequence` protocol message to the RM destination. This contains a reference for the source endpoint that receives acknowledgements (`wsrm:AcksTo` endpoint).
2. The RM destination sends a `CreateSequenceResponse` protocol message back to the RM source. This contains the sequence ID for the RM sequence session.
3. The RM source adds an RM `Sequence` header to each message sent by the application source. This contains the sequence ID, and a unique message ID.
4. The RM source transmits each message to the RM destination.
5. The RM destination acknowledges the receipt of the message from the RM source by sending messages that contain the RM `SequenceAcknowledgement` header.
6. The RM destination delivers the message to the application destination in an exactly-once-in-order fashion.
7. The RM source retransmits a message for which it has not yet received an acknowledgement.

   The first retransmission attempt is made after a base retransmission interval. Successive retransmission attempts are made, by default, at exponential back-off intervals or, alternatively, at fixed intervals. For more details, see "Configuring WS-RM" on page 79.

This entire process occurs symmetrically for both the request and the response message; that is, in the case of the response message, the server acts as the RM source and the client acts as the RM destination.

WS-RM delivery assurances

WS-RM guarantees reliable message delivery in a distributed environment, regardless of the transport protocol used. The source or destination endpoint logs an error if reliable delivery can not be assured.

Supported specifications

Artix supports the 2005/02 version of the WS-RM specification, which is based on the WS-Addressing 2004/08 specification.

Further information

For detailed information on WS-RM, see the specification at:
http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf

WS-RM Interceptors

Overview

In Artix Java, WS-RM functionality is implemented as interceptors. The Artix Java runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it does as little work as possible, creates a message object and sends that message through an interceptor chain. Each interceptor has an opportunity to do what is required to the message. This can include reading it, transforming it, validating the message, processing headers, and so on.

Artix Java WS-RM Interceptors

The Artix Java WS-RM implementation consists of four interceptors, which are described in Table 5:

Table 5: *Artix Java WS-ReliableMessaging Interceptors*

Interceptor	Description
`org.apache.cxf.ws.rm.RMOutInterceptor`	Deals with the logical aspects of providing reliability guarantees for outgoing messages. Responsible for sending the `CreateSequence` requests and waiting for their `CreateSequenceResponse` responses. Also responsible for aggregating the sequence properties—ID and message number—for an application message.
`org.apache.cxf.ws.rm.RMInInterceptor`	Responsible for intercepting and processing RM protocol messages and `SequenceAcknowledgement` messages that are piggybacked on application messages.
`org.apache.cxf.ws.rm.soap.RMSoapInterceptor`	Responsible for encoding and decoding the reliability properties as SOAP headers.
`org.apache.cxf.ws.rm.RetransmissionInterceptor`	Responsible for creating copies of application messages for future resending.

Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, upon intercepting the first application message on the outbound interceptor chain, the `RMOutInterceptor` sends a `CreateSequence` request and only processes the original application message after it receives the `CreateSequenceResponse` response. In addition, the WS-RM interceptors add the sequence headers to the application messages and, on the destination side, extract them from the messages. You do not, therefore, have to make any changes to your application code to make the exchange of messages reliable.

For more information on how to enable WS-RM, see "Enabling WS-RM" on page 74.

Configuring WS-RM Attributes

You can control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Artix attempts to maximize the lifetime of a sequence, thus reducing the overhead incurred by the out-of-band WS-RM protocol messages. You can, however, enforce the use of a separate sequence per application message by configuring the WS-RM source's sequence termination policy (setting the maximum sequence length to `1`).

For more information on configuring WS-RM behavior, see "Configuring WS-RM" on page 79.

Enabling WS-RM

Overview

To enable WS-RM, and thereby make the exchange of messages between two endpoints reliable, the WS-RM interceptors must be present on the interceptor chains for inbound and outbound messages and faults. WS-RM uses WS-Addressing and, therefore, the WS-Addressing interceptors must also be present on the interceptor chains.

You can ensure the presence of these interceptors in one of two ways:

- Explicitly, by adding them to the dispatch chains using Spring beans; or
- Implicitly, using WS-Policy assertions, which cause the Artix runtime to transparently add the interceptors on your behalf.

Both of these approaches to enabling WS-RM are discussed in the following subsections:

- Spring beans—explicitly adding interceptors
- Artix Java WS-Policy framework—implicitly adding interceptors

Spring beans—explicitly adding interceptors

You can enable WS-RM by adding the WS-RM and WS-Addressing interceptors to the Artix Java bus or to a consumer or service endpoint using Spring bean configuration. This is the approach taken in the WS-RM sample that is contained in the *ArtixInstallDir*`/java/samples/ws_rm` directory. The configuration file, `ws-rm.cxf`, shows the WS-RM and WS-Addressing interceptors being added one-by-one as Spring beans (see Example 10).

Example 10: *Enabling WS-RM Using Spring Beans*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
       beans
   http://www.springframework.org/schema/beans/spring-beans.xsd"
   >

   <bean id="mapAggregator"
   class="org.apache.cxf.ws.addressing.MAPAggregator"/>
```

Example 10: *Enabling WS-RM Using Spring Beans*

```
    <bean id="mapCodec"
    class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
3   <bean id="rmLogicalOut"
    class="org.apache.cxf.ws.rm.RMOutInterceptor">
        <property name="bus" ref="cxf"/>
    </bean>
    <bean id="rmLogicalIn"
    class="org.apache.cxf.ws.rm.RMInInterceptor">
        <property name="bus" ref="cxf"/>
    </bean>
    <bean id="rmCodec"
    class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>

    <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
4       <property name="inInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
5       <property name="inFaultInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
6       <property name="outInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
7       <property name="outFaultInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
```

Example 10: *Enabling WS-RM Using Spring Beans*

```
        </property>
    </bean>
</beans>
```

The code shown in Example 10 on page 74 can be explained as follows:

1. An Artix Java configuration file is really a Spring XML file. You must include an opening Spring `<beans>` element that declares the namespaces and schema files for the child elements that are encapsulated by the `<beans>` element.
2. Configures each of the WS-Addressing interceptors—`MAPAggregator` and `MAPCodec`.
3. Configures each of the WS-RM interceptors—`RMOutInterceptor`, `RMInInterceptor`, and `RMSoapInterceptor`.
4. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.
5. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.
6. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.
7. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound faults.

Artix Java WS-Policy framework—implicitly adding interceptors

The Artix Java WS-Policy framework provides the infrastructure and APIs that allow you to use WS-Policy. It is compliant with the November 2006 draft publications of the Web Services Policy 1.5—Framework and Web Services Policy 1.5—Attachment specifications.

To enable WS-RM using the Artix Java WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. Example 11 shows a reference bean nested within a `jaxws:feature` element. The reference bean specifies the `AddressingPolicy`, which is defined as a separate element within the same configuration file.

Example 11: *Configuring WS-RM using WS-Policy*

```
<jaxws:client>
    <jaxws:features>
      <ref bean="AddressingPolicy"/>
    </jaxws:features>
</jaxws:client>

<wsp:Policy wsu:Id="AddressingPolicy"
   xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
    <wsam:Addressing>
      <wsp:Policy>
        <wsam:NonAnonymousResponses/>
      </wsp:Policy>
    </wsam:Addressing>
</wsp:Policy>
```

2. Add a reliable messaging policy to the `wsdl:service` element—or any other WSDL element that can be used as an attachment point for policy or policy reference elements—in your WSDL file, as shown in Example 12.

Example 12: *Adding an RM Policy to Your WSDL File*

```
<wsp:Policy wsu:Id="RM"
   xmlns:wsp="http://www.w3.org/2006/07/ws-policy"

   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-20040
   1-wss-wssecurity-utility-1.0.xsd">
```

Example 12: *Adding an RM Policy to Your WSDL File*

```
    <wsam:Addressing
   xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
    </wsam:Addressing>
    <wsrmp:RMAssertion
   xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   >
       <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
    </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
    <wsdl:port binding="tns:GreeterSOAPBinding"
   name="GreeterPort">
        <soap:address
   location="http://localhost:9020/SoapContext/GreeterPort"/>
        <wsp:PolicyReference URI="#RM"
   xmlns:wsp="http://www.w3.org/2006/07/ws-policy"/>
    </wsdl:port>
</wsdl:service>
```

Configuring WS-RM

Overview

You can configure WS-RM by:

- Setting Artix-specific attributes that are defined in the Artix Java WS-RM manager namespace, `http://cxf.apache.org/ws/rm/manager;` and
- Setting standard WS-RM policy attributes that are defined in the `http://schemas.xmlsoap.org/ws/2005/02/rm/policy` namespace.

In this section

This section describes how to set both attribute types. It includes the following subsections:

Configuring Artix-Specific WS-RM Attributes

Overview

To configure the Artix-specific attributes, use the Artix Java `rmManager` custom Spring bean. In your Artix Java configuration file:

1. Add the `http://cxf.apache.org/ws/rm/manager` namespace to your list of namespaces.
2. Add an `rmManager` custom Spring bean for the specific attribute that your want to configure, as shown in Example 13.

Example 13: *Configuring Artix-Specific WS-RM Attributes*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1      xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
       http://cxf.apache.org/ws/rm/manager
   http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
       ">
...

2  <wsrm-mgr:rmManager>
     ...Your configuration goes here
   </wsrm-mgr:rmManager>
```

Child elements of rmManager custom Spring bean

Table 6 shows the child elements of the `rmManager` custom Spring bean, defined in the `http://cxf.apache.org/ws/rm/manager` namespace:

Table 6: *Child Elements of the rmManager Custom Spring Bean*

Element	Descriptions
`RMAssertion`	An element of type `RMAssertion`.
`deliveryAssurance`	An element of type `DeliveryAssuranceType` that describes the delivery assurance that should apply.

Table 6: *Child Elements of the rmManager Custom Spring Bean*

Element	Descriptions
`sourcePolicy`	An element of type `SourcePolicyType` that allows you to configure details of the RM source.
`destinationPolicy`	An element of type `DestinationPolicyType` that allows you to configure details of the RM destination.

More detailed reference information

For more detailed reference information, including descriptions of each element's sub-elements and attributes, please refer to the Artix Configuration Reference, Java Runtime.

Example

For an example, see "Maximum unacknowledged messages threshold" on page 89.

Configuring Standard WS-RM Policy Attributes

Overview

You can configure standard WS-RM policy attributes in one of the following ways:

- RMAssertion in rmManager custom Spring bean
- Policy within a feature
- WSDL file
- External attachment

WS-Policy RMAssertion Child Elements

Table 7 shows the elements defined in the `http://schemas.xmlsoap.org/ws/2005/02/rm/policy` namespace:

Table 7: *Child Elements of the WS-Policy RMAssertion*

Name	Description
`InactivityTimeout`	Specifies the duration after which an endpoint that has received no application or control messages may consider the RM sequence to have been terminated due to inactivity.
`BaseRetransmissionInterval`	Sets the interval within which an acknowledgement must be received by the RM Source for a given message. If an acknowledgement is not received within the time set by the `BaseRetransmissionInterval`, the RM Source will retransmit the message.
`ExponentialBackoff`	Indicates the retransmission interval will be adjusted using the commonly known exponential backoff algorithm (Tanenbaum). For more information, see *Computer Networks*, Andrew S. Tanenbaum, Prentice Hall PTR, 2003.

Table 7: *Child Elements of the WS-Policy RMAssertion*

Name	Description
`AcknowledgementInterval`	In WS-RM, acknowledgements are sent on return messages or sent stand-alone. If a return message is not available to send an acknowledgement, an RM Destination can wait for up to the acknowledgement interval before sending a stand-alone acknowledgement. If there are no unacknowledged messages, the RM Destination can choose not to send an acknowledgement.

More detailed reference information

For more detailed reference information, including descriptions of each element's sub-elements and attributes, please refer to http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd.

RMAssertion in rmManager custom Spring bean

You can configure standard WS-RM policy attributes by adding an `RMAssertion` within an Artix Java `rmManager` custom Spring bean. This is a good approach if you want to keep all of your WS-RM configuration in the same configuration file; that is, if you want to configure Artix-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 14 shows:

1. A standard WS-RM policy attribute, `BaseRetransmissionInterval`, being configured using an `RMAssertion` within an `rmManager` custom Spring bean; and
2. An Artix-specific RM attribute, `intraMessageThreshold`, being configured in the same configuration file.

Example 14: *Configuring WS-RM Attributes Using an RMAssertion in an rmManager Custom Spring Bean*

```
<beans
   xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/p
   olicy"
   xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
```

Example 14: *Configuring WS-RM Attributes Using an RMAssertion in an rmManager Custom Spring Bean*

```
...>

<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
1       <wsrm-policy:RMAssertion>
            <wsrm-policy:BaseRetransmissionInterval
       Milliseconds="4000"/>
        </wsrm-policy:RMAssertion>
2       <wsrm-mgr:destinationPolicy>
            <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
        </wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>
```

Policy within a feature

You can configure standard WS-RM policy attributes within features, as shown in Example 15.

Example 15: *Configuring WS-RM Attributes as a Policy within a Feature*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:wsa="http://cxf.apache.org/ws/addressing"
        xmlns:wsp="http://www.w3.org/2006/07/ws-policy"

    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-20040
    1-wss-wssecurity-utility-1.0.xsd"
        xmlns:jaxws="http://cxf.apache.org/jaxws"
        xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy
    http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing
    http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws
    http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd"
    >

     <jaxws:endpoint
    name="{http://cxf.apache.org/greeter_control}GreeterPort"
    createdFromAPI="true">
        <jaxws:features>
```

Example 15: *Configuring WS-RM Attributes as a Policy within a Feature*

```
                <wsp:Policy>
                    <wsrm:RMAssertion
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                      <wsrm:AcknowledgementInterval
   Milliseconds="200" />
          </wsrm:RMAssertion>
                    <wsam:Addressing
   xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                       <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                       </wsp:Policy>
                    </wsam:Addressing>
               </wsp:Policy>
          </jaxws:features>
     </jaxws:endpoint>
</beans>
```

WSDL file

If you use the Artix Java WS-Policy framework to enable WS-RM, you can configure standard WS-RM policy attributes in your WSDL file. This is a good approach if you want your service to interoperate and use WS-RM seamlessly with consumers deployed to other policy-aware Web services stacks.

For an example, see "Artix Java WS-Policy framework—implicitly adding interceptors" on page 77 in which the base retransmission interval is configured in the WSDL file.

External attachment

You can configure standard WS-RM policy attributes in an external attachment file. This is a good approach if you cannot or do not want to change your WSDL file.

Example 16 shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 16: *Configuring WS-RM in an External Attachment*

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
   xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsp:PolicyAttachment>
        <wsp:AppliesTo>
            <wsa:EndpointReference>
```

Example 16: *Configuring WS-RM in an External Attachment*

```
   <wsa:Address>http://localhost:9020/SoapContext/GreeterPort</w
   sa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:Policy>
            <wsam:Addressing
   xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsp:Policy/>
            </wsam:Addressing>
            <wsrmp:RMAssertion
   xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   >
                <wsrmp:BaseRetransmissionInterval
   Milliseconds="30000"/>
            </wsrmp:RMAssertion>`
        </wsp:Policy>
    </wsp:PolicyAttachment>
</attachments>
```

WS-RM Configuration Use Cases

Overview

This subsection focuses on configuring WS-RM attributes from a use case point of view. Where an attribute is a standard WS-RM policy attribute, defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace, only the example of setting it in an `RMAssertion` within an `rmManager` custom Spring bean is shown. For details of how to set such attributes as a policy within a feature; in a WSDL file; or in an external attachment, see "Configuring Standard WS-RM Policy Attributes" on page 82.

The following use cases are covered:

- Base retransmission interval
- Exponential backoff for retransmission
- Acknowledgement interval
- Maximum unacknowledged messages threshold
- Maximum length of an RM sequence
- Message delivery assurance policies

Base retransmission interval

The `BaseRetransmissionInterval` element specifies the interval at which an RM source retransmits a message that has not yet been acknowledged. It is defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd schema file. The default value is 3000 milliseconds.

Configuring the base retransmission interval

The following example shows how to set the WS-RM base retransmission interval:

```
<beans
   xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/p
   olicy
...>

<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval
   Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>

</beans>
```

Exponential backoff for retransmission

The `ExponentialBackoff` element determines if successive retransmission attempts for an unacknowledged message are performed at exponential intervals.

The presence of the `ExponentialBackoff` element enables this feature and an exponential backoff ratio of `2` is used by default.

Configuring exponential backoff for retransmission

The following example shows how to set the WS-RM exponential backoff for retransmission:

```
<beans
   xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/p
   olicy
...>

<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff="4"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>

</beans>
```

Acknowledgement interval

The `AcknowledgementInterval` element specifies the interval at which the WS-RM destination sends asynchronous acknowledgements. These are in addition to the synchronous acknowledgements that it sends upon receipt of an incoming message. The default asynchronous acknowledgement interval is `0` milliseconds. This means that if the `AcknowledgementInterval` is not configured to a specific value, acknowledgements are sent immediately (that is, at the first available opportunity).

Asynchronous acknowledgements are sent by the RM destination only if both of the following conditions are met:

- The RM destination is using a non-anonymous `wsrm:acksTo` endpoint.
- The opportunity to piggyback an acknowledgement on a response message does not occur before the expiry of the acknowledgement interval.

Configuring the WS-RM acknowledgement interval

The following example shows how to set the WS-RM acknowledgement interval:

```
<beans
   xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/p
   olicy
...>

<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval
   Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>

</beans>
```

Maximum unacknowledged messages threshold

The `maxUnacknowledged` attribute sets the maximum number of unacknowledged messages that can accrue per sequence before the sequence is terminated.

Configuring the maximum unacknowledged messages threshold

The following example shows how to set the WS-RM maximum unacknowledged messages threshold:

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>

<wsrm-mgr:rmManager>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy
   maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:rmManager>

</beans>
```

Maximum length of an RM sequence

The `maxLength` attribute sets the maximum length of a WS-RM sequence. The default value is `0`, which means that the length of a WS-RM sequence is unbound.

When this attribute is set, the RM endpoint creates a new RM sequence when the limit is reached and after receiving all of the acknowledgements for the previously sent messages. The new message is sent using a new sequence.

Configuring the maximum length of a WS-RM sequence

The following example shows how to set the maximum length of an RM sequence:

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>

<wsrm-mgr:rmManager>
    <wsrm-mgr:sourcePolicy>
       <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
   </wsrm-mgr:sourcePolicy>
</wsrm-mgr:rmManager>

</beans>
```

Message delivery assurance policies

You can configure the RM destination to use the following delivery assurance policies:

- `AtMostOnce`—The RM destination delivers the messages to the application destination at most once without duplication or an error will be raised. It is possible that some messages in a sequence may not be delivered.
- `AtLeastOnce`—The RM destination delivers the messages to the application destination at least once. Every message sent will be delivered or an error will be raised. Some messages might be delivered more than once.
- `InOrder`—The RM destination delivers the messages to the application destination in the order that they are sent. This delivery assurance can be combined with the `AtMostOnce` or `AtLeastOnce` assurances.

Configuring WS-RM message delivery assurance policies

The following example show how to set the WS-RM message delivery assurance:

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>

<wsrm-mgr:rmManager>
    <wsrm-mgr:deliveryAssurance>
        <wsrm-mgr:AtLeastOnce />
    </wsrm-mgr:deliveryAssurance>
</wsrm-mgr:rmManager>

</beans>
```

Configuring WS-RM Persistence

Overview

The Artix WS-RM features already described in this chapter provide reliability for cases such as network failures. WS-RM persistence provides reliability across other types of failure such as an RM source or destination crash.

WS-RM persistence involves storing the state of the various RM endpoints in persistent storage. This enables the endpoints when reincarnated to continue sending and receiving messages as they did before the crash.

Artix enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Artix includes Derby for out-of-the-box deployment. In addition, the persistent store is also exposed using a Java API. If you want to implement your own persistence mechanism, you can implement one using this API with your preferred DB (see Developing Artix Applications with JAX-WS).

Note: WS-RM persistence is supported for oneway calls only. It is disabled by default.

How it works

Artix WS-RM persistence works as follows:

- At the RM source endpoint, an outgoing message is persisted before transmission. It is evicted from the persistent store after the acknowledgement is received.
- After a recovery from crash, it recovers the persisted messages and retransmits until all the messages have been acknowledged. At that point, the RM sequence is closed.
- At the RM destination endpoint, an incoming message is persisted, and upon a successful store, the acknowledgement is sent. When a message is successfully dispatched, it is evicted from the persistent store.
- After a recovery from a crash, it recovers the persisted messages and dispatches them. It also brings the RM sequence to a state where new messages are accepted, acknowledged, and delivered.

Enabling WS-persistence

To enable WS-RM persistence, you must specify the object implementing the persistent store for WS-RM. You can develop your own or use the JDBC based store that comes with Artix.

The configuration shown below enables the JDBC-based store that comes with Artix:

```
<bean id="RMTxStore"
   class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>

<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

Configuring WS-persistence

The JDBC-based store that comes with Artix supports the properties shown in Table 8:

Table 8: *JDBC Store Properties*

Attribute Name	Type	Default Setting
`driverClassName`	`String`	`org.apache.derby.jdbc.EmbeddedDriver`
`userName`	`String`	`null`
`passWord`	`String`	`null`
`url`	`String`	`jdbc:derby:rmdb;create=true`

The configuration below shows an example of enabling the JDBC-based store that comes with Artix, while setting the `driverClassName` and `url` to non-default values:

```
<bean id="RMTxStore"
   class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName"
   value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

CHAPTER 7

Publishing WSDL Contracts

This chapter describes how to publish WSDL files that correspond to specific Web services. This enables consumers to access a WSDL file and invoke on a service.

In this chapter

This chapter discusses the following topics:

Artix WSDL Publishing Service

Overview

The Artix WSDL publishing service enables Artix processes to publish WSDL files for specific Web services. Published WSDL files can be downloaded by consumers or viewed in a Web browser. They can also be downloaded by Web service processes created by other vendor tools (for example, Systinet).

The WSDL publishing service enables Artix applications to be used in various deployment models—for example, J2EE—without the need to specify file system locations. It is the recommended way to publish WSDL files for Artix services.

The WSDL publishing service is implemented by the `com.iona.cxf.wsdlpublish.WSDLPublish` class. This class can be loaded by any Artix process that hosts a Web service endpoint. This includes server applications, Artix routing applications, and applications that expose a callback object.

Use with endpoint references

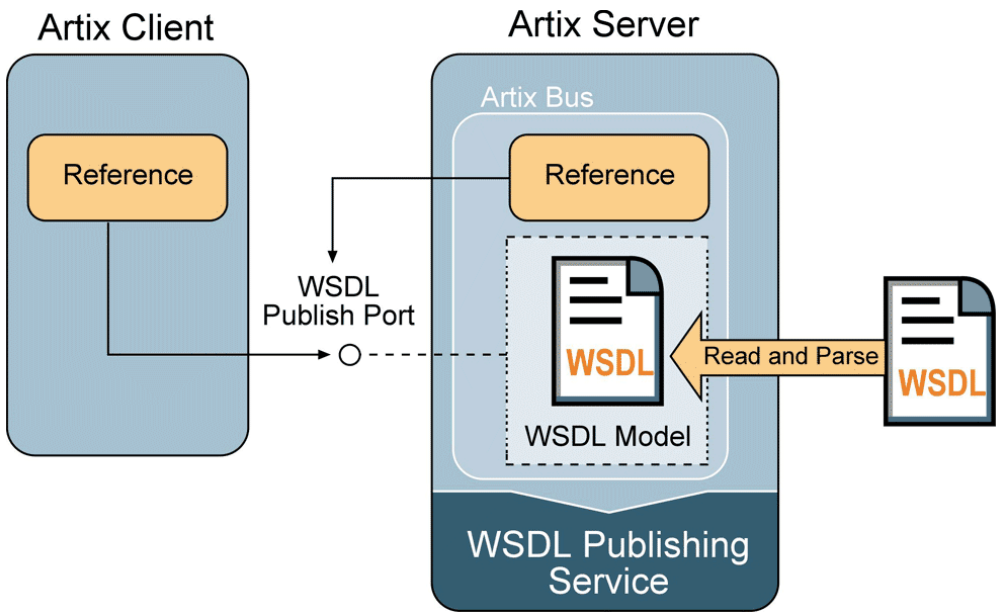
It is recommended that you use the WSDL publishing service for any applications that generate and export references. To use references, the consumer must have access to the WSDL file referred to by the reference. The simplest way to accomplish this is to use the WSDL publishing service.

Figure 5 shows an example of creating references with the WSDL publishing service. The WSDL publishing service automatically opens a port, from which consumers can download a copy of the server's dynamically updated WSDL file. Generated references have their WSDL location set to the following URL:

```
http://Hostname:WSDLPublishPort/QueryString
```

Hostname is the server host, *WSDLPublishPort* is a TCP/IP port used to serve up WSDL file, and *QueryString* is a string that requests a particular WSDL file (see "Querying the WSDL Publishing Service" on page 102). If a client accesses the WSDL location URL, the server converts the WSDL model to XML on the fly and returns the WSDL contract in a HTTP message.

Figure 5: *Creating References with the WSDL Publishing Service*



Multiple transports

The WSDL publishing service makes the WSDL file available through an HTTP URL. However, the Web service described in the WSDL file can use a transport other than HTTP.

Configuring the WSDL Publishing Service

Overview

To configure the WSDL publishing service in the Artix Java runtime you must create an Artix Java configuration file to set the configuration options that are described in this section.

Configuration file

Example 17 shows an example of such a configuration file. It is written using plain Spring beans. For more detailed information on each of the configuration options, see "WSDL publishing service configuration options" on page 100:

Example 17: *Configuring the WSDL Publishing Service*

```
1  <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
       beans http://www.springframework.org/schema/beans/
       spring-beans-2.0.xsd">

2      <bean id="WSDLPublishManager"
       class="com.iona.cxf.wsdlpublish.WSDLPublishManager">
          <property name="enabled" value="true"/>
          <property name="bus" ref="cxf"/>
          <property name="WSDLPublish" ref="WSDLPublish"/>
       </bean>

3      <bean id="WSDLPublish"
       class="com.iona.cxf.wsdlpublish.WSDLPublish">
4         <property name="publishPort" value="27220"/>
5         <property name="publishHostname" value="myhost"/>
6         <property name="catalogFacility" value="true"/>
7         <property name="processWSDL" value="standard"/>
8         <property name="removeSchemas" ref="rschemas"/>
       </bean>
9      <bean id="rschemas"
       class="com.iona.cxf.wsdlpublish.Valuelist"
       value="http://cxf.apache.org/ http://schemas.iona.com/"/>

   </beans>
```

The configuration shown in Example 17 can be explained as follows:

1. An Artix Java configuration file is really a Spring XML file. You must include an opening Spring `<beans>` element that declares the namespaces and schema files for the child elements that are encapsulated by the `<beans>` element.
2. Specifies the `com.iona.cxf.wsdlpublish.WSDLPublishManager` class, which implements the WSDL publishing service manager. The WSDL publishing service manager enables the WSDL publishing service.
3. Specifies the `com.iona.cxf.wsdlpublish.WSDLPublish` class, which implements the WSDL publishing service.
4. The `publishPort` property specifies the TCP/IP port on which the WSDL files are published.
5. The `publishHostname` property specifies the hostname on which the WSDL publishing service is available.
6. The `catalogFacility` property specifies that the catalog facility is enabled.
7. The `processWSDL` property specifies the type of processing that is done on the WSDL file before the WSDL file is published.
8. The `removeSchemas` property specifies a list of the target namespaces of the extensions that are removed when the `processWSDL` property is set to `standard`. It this example it references `rschemas`, which is configured in the next line of code. See **9** below.
9. Configures a `rschema` bean, which specifies the `com.iona.cxf.wsdlpublish.Valuelist` class. The `com.iona.cxf.wsdlpublish.Valuelist` class has a `value` attribute, which you can use to list the schemas that you want removed from the WSDL file. In this case, `http://cxf.apache.org/` and `http://schemas.iona.com/` are removed.

WSDL publishing service configuration options

Table 9 describes each of the WSDL publishing service configuration options.

Table 9: *WSDL Publishing Service Configuration Options*

Configuration Option	Description
`publishPort`	An integer that specifies the TCP/IP port that WSDL files are published on. If the port is in use, the server process will start and an error message indicating the address is already in use will be raised. The default value is `27220`.
`publishHostname`	A string that specifies the hostname on which the WSDL publishing service is available. The default value is `localhost`.
`catalogFacility`	A boolean that when set to `true` enables the catalog facility, and when set to `false` disables the catalog facility. A catalog facility provides another way to access WSDL and `.xsd` files (as opposed to on a file system). The default value is `true`.

Table 9: *WSDL Publishing Service Configuration Options*

Configuration Option	Description
`processWSDL`	A string that specifies the type of processing that is done on the WSDL file before the WSDL file is published. The `processWSDL` option has three possible values: • `none`—no processing of the WSDL file takes place; that is, the WSDL document is published as is. • `artix`—the WSDL file is processed so that relative paths of imported/included schemas are modified, and the imported/included schemas are published on the modified path. • `standard`—same as `artix`, but non-standard extensions are also removed. The default setting is `artix`.
`removeSchemas`	A valuelist that removes the target namespaces that are listed when the `processWSDL` option is set to `standard`. The default setting is `http://cxf.apache.org/` and `http://schemas.iona.com/`.

Querying the WSDL Publishing Service

Overview

Each HTTP `GET` request for a WSDL file must have a query appended to it. The Artix Java runtime supports RESTful services and, as a result, an HTTP `GET` request is not automatically destined for the WSDL publishing service.

The WSDL publishing service supports the following queries:

`?wsdl`	Returns the WSDL file for the published endpoint.
`?xsd`	Returns the schema file for the published endpoint.
`?services`	Returns a HTML formatted page with a list of all published endpoints and any resolved schemas.

Example query syntax

The following are examples of query syntax that are serviced:

- Using `?wsdl`:

```
http://localhost:27220/SoapContext2/SoapPort2?wsdl
```

- Using `?xsd`. If a WSDL file has an imported schema, for example, `schema1.xsd`, you can find the schema using the following query:

```
http://localhost:27220/SoapContext2/SoapPort2?xsd=schema1.xsd
```

- Using `?services`:

```
http://localhost:27220?services
```

Returns a HTTP page that lists all documents associated with active services.

CHAPTER 8

Enabling High Availability

This chapter explains how to enable and configure high availability (HA) in the Artix Java runtime.

In this chapter

This chapter discusses the following topics:

Introduction to High Availability

Overview

Scalable and reliable Artix applications require high availability to avoid any single point of failure in a distributed system. You can protect your system from single points of failure using *replicated services*.

A replicated service is comprised of multiple instances, or *replicas*, of the same service. Together these act as a single logical service. Clients invoke requests on the replicated service, and Artix delivers the requests to one of the member replicas. The routing to a replica is transparent to the client.

HA with static failover

Artix supports HA with static failover in which replica details are encoded in the service WSDL file. The WSDL file contains multiple ports, and possibly multiple hosts, for the same service. The number of replicas in the cluster remains static as long as the WSDL file remains unchanged. Changing the cluster size involves editing the WSDL file.

HA with dynamic failover

Artix also supports HA with dynamic failover. HA with dynamic failover is one in which number of replicas in a cluster can be dynamically increased and decreased simply by starting and stopping instances of the server application. The Artix locator service is central to this feature.

The Artix locator service provides a lightweight mechanism for balancing workloads among a group of services. When several services with the same service name register with the Artix locator service, it automatically creates a list of references to each instance of this service. The locator hands out references to clients using a round-robin or random algorithm. This process is automatic and invisible to both clients and services.

The discovery mechanism can also be used in failover scenarios. The Artix locator service only hands out references for service replicas that it believes to be active, on the basis of the dynamic state of the cluster as maintained by the peer manager instance collocated with the Artix locator service. Mutual heart-beating between the peer manager instances associated with the Artix locator service and service replicas, allow each to detect the availability of the other.

Dynamic failover also has the advantage that cluster membership is not fixed. It is easy to grow or shrink the cluster size by simply starting and stopping replica instances. Newly started replicas transparently register with the Artix locator service, and their references are immediately eligible for discovery by new clients. Similarly, gracefully shutdown services transparently deregister themselves with the Artix locator service.

Sample applications

The examples shown in this chapter are taken from the HA sample applications that are located in the following directory of your Artix installation:

```
ArtixInstallDir/java/samples/ha
```

For information on how to run these samples applications, see the `README.txt` files on the sample directories.

More information about the locator service

For more information on the Artix locator service, including how to configure it, see the Artix Locator Guide.

Enabling HA with Static Failover

Overview

To enable HA with static failover, you must:

- Encode replica details in your service WSDL file
- Add the clustering feature to your client configuration

Encode replica details in your service WSDL file

You must encode the details of the replicas in your cluster in your service WSDL file. Example 18 shows a WSDL file extract that defines a service cluster of three replicas:

Example 18: *Enabling HA with Static Failover—WSDL File*

```
1   <wsdl:service name="ClusteredService">
        <wsdl:port binding="tns:Greeter_SOAPBinding"
i       name="Replica1">
            <soap:address
        location="http://localhost:9001/SoapContext/Replica1"/>
        </wsdl:port>

ii      <wsdl:port binding="tns:Greeter_SOAPBinding"
        name="Replica2">
            <soap:address
        location="http://localhost:9002/SoapContext/Replica2"/>
        </wsdl:port>

iii     <wsdl:port binding="tns:Greeter_SOAPBinding"
        name="Replica3">
            <soap:address
        location="http://localhost:9003/SoapContext/Replica3"/>
        </wsdl:port>

    </wsdl:service>
```

The WSDL extract shown in Example 18 on page 106 is taken from the `replicated_hello_world.wsdl` file located in the *`ArtixInstallDir`*`/java/samples/ha/static_failover/wsdl` directory. It can be explained as follows:

1. Defines a service, `ClusterService`, which is exposed on three ports:
   i. `Replica1`—exposes the `ClusterService` as a SOAP over HTTP endpoint on port `9001`.
   ii. `Replica2`—exposes the `ClusterService` as a SOAP over HTTP endpoint on port `9002`.
   iii. `Replica3`—exposes the `ClusterService` as a SOAP over HTTP endpoint on port `9003`.

Add the clustering feature to your client configuration

In your client configuration file, add the clustering feature as shown in Example 19:

Example 19: *Enabling HA with Static Failover—Client Configuration*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:clustering="http://cxf.apache.org/clustering"
       xsi:schemaLocation="
http://cxf.apache.org/jaxws
   http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd"
   >

    <jaxws:client
   name="{http://apache.org/hello_world_soap_http}Replica1"
                createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client
   name="{http://apache.org/hello_world_soap_http}Replica2"
                createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
```

Example 19: *Enabling HA with Static Failover—Client Configuration*

```
        </jaxws:features>
    </jaxws:client>

    <jaxws:client
   name="{http://apache.org/hello_world_soap_http}Replica3"
                createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

</beans>
```

Configuring HA with Static Failover

Overview

By default, HA with static failover uses a sequential strategy when selecting a replica service if the original service with which a client is communicating becomes unavailable or fails. The sequential strategy selects a replica service in the same sequential order every time it is used. Selection is determined by Artix' internal service model and results in a deterministic failover pattern.

Configuring a random strategy

You can configure HA with static failover to use a random strategy instead of the sequential strategy when selecting a replica. The random strategy selects a replica service at random each time a service becomes unavailable or fails. The choice of failover target from the surviving members in a cluster is entirely random.

To configure the random strategy, adding the configuration shown in Example 20 to your client configuration file:

Example 20: *Configuring a Random Strategy for Static Failover*

```
<beans ...>
1       <bean id="Random"
        class="org.apache.cxf.clustering.RandomStrategy"/>

        <jaxws:client
        name="{http://apache.org/hello_world_soap_http}Replica3"
                      createdFromAPI="true">
            <jaxws:features>
                <clustering:failover>
2                   <clustering:strategy>
                        <ref bean="Random"/>
                    </clustering:strategy>
                </clustering:failover>
            </jaxws:features>
        </jaxws:client>
</beans>
```

The configuration shown in Example 20 on page 109 can be explained as follows:

1. Defines a `Random` bean and implementation class that implements the random strategy.
2. Specifies that the random strategy be used when selecting a replica.

Enabling HA with Dynamic Failover

Overview

To enable HA with dynamic failover, you must:

1. Configure your service to register with the Artix locator
2. Configure your client to use locator meditated failover
3. Ensure the Artix locator is running

Configure your service to register with the Artix locator

To configure your service to register with the Artix locator service add configuration shown in Example 21 to your server configuration file.

Example 21: *Configuring Your Service to Register with the Locator*

```
<beans xmlns="http://www.springframework.org/schema/beans"

   xmlns:locatorEndpoint="http://cxf.iona.com/locator/endpoint"
...>

1    <bean id="LocatorSupport"
   class="com.iona.cxf.locator.LocatorSupport">
        <property name="bus" ref="cxf"/>
        <property name="contract">
            <value>./wsdl/locator.wsdl</value>
        </property>
    </bean>

    <jaxws:endpoint
   name="{http://apache.org/hello_world_soap_http}SoapPort"
                    createdFromAPI="true">
2       <jaxws:features>
            <locatorEndpoint:registerOnPublish
   monitorLiveness="true"/>
        </jaxws:features>
    </jaxws:endpoint>

</beans>
```

The configuration shown in Example 21 is taken from the HA sample and can be explained as follows:

1. Enables the service to use the Artix locator service.
2. The `registerOnPublish` feature enables the published endpoint to register with the Artix locator service.

Configure your client to use locator meditated failover

To configure your client to use locator mediated failover add the configuration shown in Example 22 to your client configuration file.

Example 22: *Configuring your Client to Use Locator Mediated Failover*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:clustering="http://cxf.apache.org/clustering"...>

1      <bean id="LocatorSupport"
    class="com.iona.cxf.locator.LocatorSupport">
        <property name="bus" ref="cxf"/>
        <property name="contract">
            <value>./wsdl/locator.wsdl</value>
        </property>
    </bean>

2      <bean id="LocatorMediated"

    class="com.iona.cxf.ha.failover.LocatorMediatedStrategy">
        <property name="bus" ref="cxf"/>
    </bean>

    <jaxws:client
    name="{http://apache.org/hello_world_soap_http}SoapPort"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover>
                <clustering:strategy>
                    <ref bean="LocatorMediated"/>
                </clustering:strategy>
            </clustering:failover>
        </jaxws:features>
    </jaxws:client>

</beans>
```

The configuration shown in Example 22 on page 112 is from the HA sample and can be explained as follows:

1. Enables the client to use the Artix locator service to find services.
2. Enables failover support using the Artix locator service.

Ensure the Artix locator is running

Ensure that the Artix locator service is running. To start the Artix locator service, run the following command:

```
ArtixInstallDir/java/bin/start_locator.bat
```

For more information, see the Artix Locator Guide.

Configuring HA with Dynamic Failover

Overview

You can change the default behavior of HA with dynamic failover by configuring the following aspects of the feature:

- Enabling Artix locator to check the state of a registered service
- Setting the heartbeat interval
- Initial delay in locator response
- Maximum number of client retries
- Delay between client retry attempts
- Sequential backoff in client retry attempts

Enabling Artix locator to check the state of a registered service

The `monitorLiveness` attribute enables the Artix locator service to check, at regular intervals, whether a registered service is still live or not. It is disabled by default.

To enable the Artix locator service to monitor the state of a registered service, add the following to your server configuration file:

```
<locatorEndpoint:registerOnPublish monitorLiveness="true">
```

Setting the heartbeat interval

The `heartbeatInterval` attribute specifies the frequency, in milliseconds, at which the Artix locator service checks the state of a registered service. It depends on the `monitorLiveness` attribute being set to `true`. The default value is `10000` milliseconds (`10` seconds).

To change the default heartbeat interval, add the following to your server configuration file:

```
<locatorEndpoint:registerOnPublish monitorLiveness="true"
   heartbeatInterval="10001"/>
```

Initial delay in locator response

The `initialDelay` attribute specifies an initial delay, in milliseconds, in the Artix locator service's response to the client's request for an EPR. The default value is `0`.

To change the initial delay in the Artix locator's response to the client's request for an EPR, add the following to your client configuration file:

```
<bean id="LocatorMediated"
   class="com.iona.cxf.ha.failover.LocatorMediatedStrategy">
     <property name="initialDelay" value="500"/>
</bean>
```

Maximum number of client retries

The `maxRetries` attribute specifies the maximum number of times that the client retries to connect to a service. The default value is `3`.

To change the number of times that the client retries to connect to a service, add the following to your client configuration file:

```
<bean id="LocatorMediated"
   class="com.iona.cxf.ha.failover.LocatorMediatedStrategy">
     <property name="maxRetries" value="5"/>
</bean>
```

Delay between client retry attempts

The `intraRetryDelay` attribute specifies the delay, in milliseconds, between the client's attempts to retry connecting to the service. The default value is `5000` milliseconds.

To change the delay between a client's attempts to retry connecting to a service, add the following to your client configuration file:

```
<bean id="LocatorMediated"
   class="com.iona.cxf.ha.failover.LocatorMediatedStrategy">
     <property name="intraRetryDelay" value="4000"/>
</bean>
```

Sequential backoff in client retry attempts

The `backoff` attribute specifies an exponential backoff in the client's retry attempts. The default value is `1.0`, which essentially does not exponentially increase the amount of time between a client's retry attempts.

To change the exponential backoff, add the following to your client configuration file:

```
<bean id="LocatorMediated"
   class="com.iona.cxf.ha.failover.LocatorMediatedStrategy">
      <property name="backoff" value="1.2"/>
</bean>
```

Index

U

W