MICRO FOCUS

Artix 5.6.4

Deployment Guide: Java

2017-02-23

Contents

Preface

What is Covered in This Book

This book explains how to configure and deploy Artix for Java Runtime services and applications, including those written in JAX-WS and JavaScript.

Who Should Read This Book

The main audience of this book 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:

- *Artix Configuration Overview* describes Artix Java configuration files.

- *Setting Up Your Environment* describes how to set up your Artix Java environment.

- *Configuring Artix Endpoints* describes how to configure Artix Java endpoints.

- *Artix Logging* describes how to use logging.

- *Deploying to a Servlet Container* describes how to deploy an Artix Java endpoint to a servlet container.

- *Deploying WS-Addressing* describes how to configure Artix Java endpoints to use WS-Addressing.

- *Enabling Reliable Messaging* describes how to enable and configure Web Services Reliable Messaging (WS-RM).

- *Enabling High Availability* describes how to enable and configure both static failover and dynamic failover.

- *Publishing WSDL Contracts* describes how to enable the Artix Java WSDL publishing service.

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: Java***.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.

- The Knowledge Base, a large collection of product tips and workarounds.

- Examples and Utilities, including demos and additional product documentation.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.

- Your computer make and model.

- Your operating system version number and details of any networking software you are using.

- The amount of memory in your computer.

- The relevant page reference or section in the documentation.

- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)

- https://supportline.microfocus.com/productdoc.aspx (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Artix for Java Configuration Overview

Artix takes a minimalist approach to requiring configuration. However, it provides a large number of options for providing configuration data.

Artix adopts an approach of zero configuration, or configuration by exception. Configuration is required only if you want to either customize the runtime to exhibit non-default behavior or if you want to activate some of the more advanced features.

Artix for Java supports a number of configuration methods if you want to change the default behavior, enable specific functionality or fine-tune a component's behavior. The supported configuration methods include:

*   Spring XML configuration

*   WS-Policy statements

*   WSDL extensions

Spring XML configuration is, however, the most versatile way to configure Artix and is the recommended approach to use.

Artix Configuration Files

Artix for Java leverages the Spring framework to inject configuration information into the runtime when it starts up. The XML configuration file used to configure applications is a Spring XML file that contains some Artix specific elements.

Spring framework

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

For more information on the Spring framework, see http://www.springframework.org. Of particular relevance is Chapter 3 of the Spring reference guide, *The IoC container*.

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

Configuration namespace

The core Artix configuration elements are defined in the
http://cxf.apache.org/jaxws namespace. You must add the
entry shown in Example 1 to the `beans` element of your
configuration file.

Example 1. Namespace

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws
  ...>
```

Advanced features, like WS-Addressing and WS-RM, require the
use of elements in other namespaces. The SOAP and JMS
transports also use elements defined in different namespaces.
You must add those namespaces when configuring those
features.

Schema location

Spring XML files use the `beans` element's `xsi:schemaLocation`
attribute to locate the schemas required to validate the
elements used in the document. The `xsi:schemaLocation` attribute
is a list of namespaces, and the schema in which the namespace
is defined. Each namespace/schema combination is defined as
a space delimited pair.

You should add the Artix configuration schemas to the list of
schemas in the attribute as shown in Example 2.

Example 2. Adding the JAX-WS Schema to the Configuration File

```
<beans ...
   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
...">
```

Sample configuration file

Example 3 shows a simplified example of an Artix configuration
file.

Example 3. Artix Configuration File

```
<beans xmlns="http://www.springframework.org/schema/beans" ❶
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxws="http://cxf.apache.org/jaxws" ❷

  ...
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd

    http://cxf.apache.org/jaxws ❸

    http://cxf.apache.org/schemas/jaxws.xsd">

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

</beans>
```

The following describes Example 3:

❶ An Artix configuration file is actually 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.

❷ Before using the Artix configuration elements, you must declare its namespace in the configuration's root element.

❸ In order for the runtime and the tooling to ensure that your configuration file is valid, you need to add the proper entries to the schema location list.

❹ The contents of the configuration depends on the behavior you want exhibited by the runtime. You can use:

- Artix specific elements

- Plain Spring XML `bean` elements

Making Your Configuration File Available

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

- Name the configuration file `cxf.xml` and add it your `CLASSPATH`.

- Use one of the following command-line flags to point to the configuration file:

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

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

  This allows you to save the configuration file anywhere on your system and avoid adding it to your `CLASSPATH`. It also means you can give your configuration file any name you want.

This is a useful approach for portable JAX-WS applications. It is also the method used in most of the Artix samples. For example, in the WS-Addressing sample, located in the `InstallDir/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
```

NOTE: 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. For example:

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

Setting Up Your Environment

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

Using the Artix Environment Script

To use the Artix runtime environment, the host computer must have several environment variables set. Set these variables using the **artix_java_env** script.

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 *InstallDir*/bin folder.

Artix for Java Environment Variables

This section describes the following environment variables in more detail:

- ARTIX_JAVA_HOME

- CXF_HOME

- JAVA_HOME

- ANT_HOME

- PATH

- CAMEL_HOME

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

Environment variables

The environment variables are explained in Table 1.

Table 1. Artix Environment Variables

Variable	Description
ARTIX_JAVA_HOME	Specifies the top level of your Artix installation. For example, on Windows, if you install Artix into the `C:\Artix` directory, `ARTIX_JAVA_HOME` should be set to `C:\Artix`.
CXF_HOME	Specifies the top level of your Artix installation. For example, on Windows, if you install Artix into the `C:\Artix` directory, `CXF_HOME` should be set to `C:\Artix`.
JAVA_HOME	Specifies the directory path to your system JDK.
ANT_HOME	Specifies the directory path to the **ant** utility. The default location is `InstallDir\tools\ant`.
CAMEL_HOME	Specifies the directory path of the Apache Camel libraries. The default location is `InstallDir\lib\camel`.
PATH	The Artix bin directories are prepended on the PATH to ensure that the proper libraries, configuration files, and utility programs are used.

Customizing your Environment Script

The **artix_java_env** script sets the Artix environment variables using values obtained from the 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, note the points described 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

JAVA_HOME defaults to the value obtained from the installer. If you do not manually set this variable before running **artix_java_env**, it takes its value from the installer.

Configuring Artix Endpoints

Artix endpoints are configured using one of three Spring configuration elements. The correct element depends on what type of endpoint you are configuring and which features you wish to use. For consumers you use the `jaxws:client` element. For service providers you can use either the `jaxws:endpoint` element or the `jaxws:server` element.

The information used to define an endpoint is typically defined in the endpoint's contract. You can use the configuration element's to override the information in the contract. You can also use the configuration elements to provide information that is not provided in the contract.

NOTE: When dealing with endpoints developed using a Java-first approach it is likely that the SEI serving as the endpoint's contract is lacking information about the type of binding and transport to use.

You must use the configuration elements to activate advanced features such as WS-RM. This is done by providing child elements to the endpoint's configuration element.

Configuring Service Providers

Artix has two elements that can be used to configure a service provider:

- `jaxws:endpoint`

- `jaxws:server`

The differences between the two elements are largely internal to the runtime. The `jaxws:endpoint` element injects properties into the `org.apache.cxf.jaxws.EndpointImpl` object created to support a service endpoint. The `jaxws:server` element injects properties into the `org.apache.cxf.jaxws.support.JaxWsServerFactoryBean` object created to support the endpoint. The `EndpointImpl` object passes the configuration data to the `JaxWsServerFactoryBean` object. The `JaxWsServerFactoryBean` object is used to create the actual service object. Because either configuration element will configure a service endpoint, you can choose based on the syntax you prefer.

Using the jaxws:endpoint Element

The `jaxws:endpoint` element is the default element for configuring JAX-WS service providers. Its attributes and children specify all of the information needed to instantiate a service provider.

Many of the attributes map to information in the service's contract. The children are used to configure interceptors and other advanced features.

Identifying the endpoint being configured

For the runtime to apply the configuration to the proper service provider, it must be able to identify it. The basic means for identifying a service provider is to specify the class that implements the endpoint. This is done using the `jaxws:endpoint` element's `implementor` attribute.

For instances where different endpoint's share a common implementation, it is possible to provide different configuration for each endpoint. There are two approaches for distinguishing a specific endpoint in configuration:

- a combination of the `serviceName` attribute and the `endpointName` attribute

   The `serviceName` attribute specifies the `wsdl:service` element defining the service's endpoint. The `endpointName` attribute specifies the specific `wsdl:port` element defining the service's endpoint. Both attributes are specified as QNames using the format *ns:name*. *ns* is the namespace of the element and *name* is the value of the element's `name` attribute.

   > **TIP:** If the `wsdl:service` element only has one `wsdl:port` element, the `endpointName` attribute can be omitted.

- the `name` attribute

   The `name` attribute specifies the QName of the specific `wsdl:port` element defining the service's endpoint. The QName is provided in the format *{ns}localPart*. *ns* is the namespace of the `wsdl:port` element and *localPart* is the value of the `wsdl:port` element's `name` attribute.

Attributes

The attributes of the `jaxws:endpoint` element configure the basic properties of the endpoint. These properties include the address of the endpoint, the class that implements the endpoint, and the `bus` that hosts the endpoint.

Table 2 describes the attribute of the `jaxws:endpoint` element.

Table 2. Attributes for Configuring a JAX-WS Service Provider Using the `jaxws:endpoint` Element

Attribute	Description
`id`	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
`implementor`	Specifies the class implementing the service. You can specify the implementation class using either the class name or an ID reference to a Spring bean configuring the implementation class. This class must be on the classpath.
`implementorClass`	Specifies the class implementing the service. This attribute is useful when the value provided to the `implementor` attribute is a reference to a bean that is wrapped using Spring AOP.
`address`	Specifies the address of an HTTP endpoint. This value overrides the value specified in the services contract.
`wsdlLocation`	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the service is deployed.
`endpointName`	Specifies the value of the service's `wsdl:port` element's `name` attribute. It is specified as a QName using the format *ns:name* where *ns* is the namespace of the `wsdl:port` element.
`serviceName`	Specifies the value of the service's `wsdl:service` element's `name` attribute. It is specified as a QName using the format *ns:name* where *ns* is the namespace of the `wsdl:service` element.
`publish`	Specifies if the service should be automatically published. If this is set to `false`, the developer must explicitly publish the endpoint as described in *Publishing a Service* in ***Developing Artix Applications with JAX-WS and JAX-RS***
`bus`	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
`bindingUri`	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Artix Binding IDs.

Attribute	Description
name	Specifies the stringified QName of the service's wsdl:port element. It is specified as a QName using the format {ns}localPart. ns is the namespace of the wsdl:port element and localPart is the value of the wsdl:port element's name attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is false. Setting this to true instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.
createdFromAPI	Specifies that the user created that bean using Artix APIs, such as Endpoint.publish() or Service.getPort(). The default is false. Setting this to true does the following: 1. Changes the internal name of the bean by appending .jaxws-endpoint to its id 2. Makes the bean abstract

In addition to the attributes listed in Table 2, you might need to use multiple xmlns:shortName attributes to declare the namespaces used by the endpointName and serviceName attributes.

Example

Example 4 shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published. The example assumes that you want to use the defaults for all other values or that the implementation has specified values in the annotations.

Example 4. Simple JAX-WS Endpoint Configuration

```
<beans
  ... xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="example"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo" />
</beans>
```

Example 5 shows the configuration for a JAX-WS endpoint whose contract contains two service definitions. In this case, you must specify which service definition to instantiate using the `serviceName` attribute.

Example 5. JAX-WS Endpoint Configuration with a Service Name

```
<beans ... xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="example2"
                  implementor="org.apache.cxf.example.DemoImpl"
                  serviceName="samp:demoService2"
                  xmlns:samp="http://org.apache.cxf/wsdl/example" />
                  </beans>
```

The `xmlns:samp` attribute specifies the namespace in which the WSDL `service` element is defined.

Using the jaxws:server Element

The `jaxws:server` element is an element for configuring JAX-WS service providers. It injects the configuration information into the `org.apache.cxf.jaxws.support.JaxWsServerFactoryBean`. This is an Artix specific object. If you are using a pure Spring approach to building your services, you will not be forced to use Artix specific APIs to interact with the service.

The attributes and children of the `jaxws:server` element specify all of the information needed to instantiate a service provider. The attributes specify the information that is required to instantiate an endpoint. The children are used to configure interceptors and other advanced features.

Identifying the endpoint being configured

In order for the runtime to apply the configuration to the proper service provider, it must be able to identify it. The basic means for identifying a service provider is to specify the class that implements the endpoint. This is done using the `jaxws:server` element's `serviceBean` attribute.

For instances where different endpoint's share a common implementation, it is possible to provide different configuration for each endpoint. There are two approaches for distinguishing a specific endpoint in configuration:

- a combination of the `serviceName` attribute and the `endpointName` attribute

  The `serviceName` attribute specifies the `wsdl:service` element defining the service's endpoint. The `endpointName` attribute specifies the specific `wsdl:port` element defining the service's

Artix Deployment Guide: Java 13

endpoint. Both attributes are specified as QNames using the format `ns:name`. `ns` is the namespace of the element and `name` is the value of the element's `name` attribute.

> **TIP:** If the `wsdl:service` element only has one `wsdl:port` element, the `endpointName` attribute can be omitted.

- the `name` attribute

  The `name` attribute specifies the QName of the specific `wsdl:port` element defining the service's endpoint. The QName is provided in the format `{ns}localPart`. `ns` is the namespace of the `wsdl:port` element and `localPart` is the value of the `wsdl:port` element's `name` attribute.

Attributes

The attributes of the `jaxws:server` element configure the basic properties of the endpoint. These properties include the address of the endpoint, the class that implements the endpoint, and the `bus` that hosts the endpoint.

Table 3 describes the attribute of the `jaxws:server` element.

Table 3.   Attributes for Configuring a JAX-WS Service Provider Using the `jaxws:server` Element

Attribute	Description
id	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
serviceBean	Specifies the class implementing the service. You can specify the implementation class using either the class name or an ID reference to a Spring bean configuring the implementation class. This class must be on the classpath.
serviceClass	Specifies the class implementing the service. This attribute is useful when the value provided to the `implementor` attribute is a reference to a bean that is wrapped using Spring AOP.
address	Specifies the address of an HTTP endpoint. This value will override the value specified in the services contract.
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the service is deployed.

Attribute	Description
endpointName	Specifies the value of the service's `wsdl:port` element's `name` attribute. It is specified as a QName using the format *ns:name*, where *ns* is the namespace of the `wsdl:port` element.
serviceName	Specifies the value of the service's `wsdl:service` element's `name` attribute. It is specified as a QName using the format *ns:name*, where *ns* is the namespace of the `wsdl:service` element.
start	Specifies if the service should be automatically published. If this is set to `false`, the developer must explicitly publish the endpoint as described in *Publishing a Service* in **Developing Artix Applications with JAX-WS and JAX-RS**.
bus	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
bindingId	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Artix Binding IDs.
name	Specifies the stringified QName of the service's `wsdl:port` element. It is specified as a QName using the format *{ns}localPart*, where *ns* is the namespace of the `wsdl:port` element and *localPart* is the value of the `wsdl:port` element's `name` attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is `false`. Setting this to `true` instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before the endpoint can be instantiated.
createdFromAPI	Specifies that the user created that bean using Artix APIs, such as `Endpoint.publish()` or `Service.getPort()`. The default is `false`. Setting this to `true` does the following: 1. Changes the internal name of the bean by appending .jaxws-endpoint to its id 2. Makes the bean abstract

In addition to the attributes listed in Table 3, you might need to use multiple `xmlns:shortName` attributes to declare the namespaces used by the `endpointName` and `serviceName` attributes.

Example

Example 6 shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published.

Example 6. Simple JAX-WS Server Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:server id="exampleServer"
                serviceBean="org.apache.cxf.example.DemoImpl"
                address="http://localhost:8080/demo" />
</beans>
```

Adding Functionality to Service Providers

The `jaxws:endpoint` and the `jaxws:server` elements provide the basic configuration information needed to instantiate a service provider. To add functionality to your service provider or to perform advanced configuration you must add child elements to the configuration.

Child elements allow you to do the following:

- Change the endpoint's logging behavior

- Add interceptors to the endpoint's messaging chain

- Enable WS-Addressing features

- Enable reliable messaging

Elements

Table 4 describes the child elements that `jaxws:endpoint` supports.

Table 4. Elements Used to Configure JAX-WS Service Providers

Element	Description
`jaxws:handlers`	Specifies a list of JAX-WS `Handler` implementations for processing messages. For more information on JAX-WS `Handler` implementations see *Writing Handlers* in ***Developing Artix Applications with JAX-WS and JAX-RS***.
`jaxws:inInterceptors`	Specifies a list of interceptors that process inbound requests.
`jaxws:inFaultInterceptors`	Specifies a list of interceptors that process inbound fault messages.
`jaxws:outInterceptors`	Specifies a list of interceptors that process outbound replies.
`jaxws:outFaultInterceptors`	Specifies a list of interceptors that process outbound fault messages.
`jaxws:binding`	Specifies a bean configuring the message binding used by the endpoint. Message bindings are configured using implementations of the `org.apache.cxf.binding.BindingFactory` interface. The SOAP binding is configured using the `soap:soapBinding` bean.
`jaxws:dataBinding`	Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition. The `jaxws:endpoint` element does not support the `jaxws:dataBinding` element.
`jaxws:executor`	Specifies a Java executor that is used for the service. This is specified using an embedded bean definition.
`jaxws:features`	Specifies a list of beans that configure advanced features of Artix for Java. You can provide either a list of bean references or a list of embedded beans.

Element	Description
`jaxws:invoker`	Specifies an implementation of the `org.apache.cxf.service.Invoker` interface used by the service. The `Invoker` implementation controls how a service is invoked. For example, it controls whether each request is handled by a new instance of the service implementation or if state is preserved across invocations.
`jaxws:properties`	Specifies a Spring map of properties that are passed along to the endpoint. These properties can be used to control features like enabling MTOM support.
`jaxws:serviceFactory`	Specifies a bean configuring the `JaxWsServiceFactoryBean` object used to instantiate the service.

Configuring Consumer Endpoints

JAX-WS consumer endpoints are configured using the `jaxws:client` element. The element's attributes provide the basic information necessary to create a consumer.

To add other functionality, like WS-RM, to the consumer you add children to the `jaxws:client` element. Child elements are also used to configure the endpoint's logging behavior and to inject other properties into the endpoint's implementation.

Basic Configuration Properties

The attributes described in Table 5 provide the basic information necessary to configure a JAX-WS consumer. You only need to provide values for the specific properties you want to configure. Most of the properties have sensible defaults, or they rely on information provided by the endpoint's contract.

Table 5. Attributes Used to Configure a JAX-WS Consumer

Attribute	Description
`address`	Specifies the HTTP address of the endpoint where the consumer will make requests. This value overrides the value set in the contract.
`bindingId`	Specifies the ID of the message binding the consumer uses. A list of valid binding IDs is provided in Artix Binding IDs.

Attribute	Description
bus	Specifies the ID of the Spring bean configuring the bus managing the endpoint.
endpointName	Specifies the value of the `wsdl:port` element's `name` attribute for the service on which the consumer is making requests. It is specified as a QName using the format `ns:name`, where `ns` is the namespace of the `wsdl:port` element.
serviceName	Specifies the value of the `wsdl:service` element's name attribute for the service on which the consumer is making requests. It is specified as a QName using the format `ns:name` where `ns` is the namespace of the `wsdl:service` element.
username	Specifies the username used for simple username/password authentication.
Password	Specifies the password used for simple username/password authentication.
serviceClass	Specifies the name of the service endpoint interface(SEI).
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the client is deployed.
Name	Specifies the stringified QName of the `wsdl:port` element for the service on which the consumer is making requests. It is specified as a QName using the format `{ns}localPart`, where `ns` is the namespace of the `wsdl:port` element and `localPart` is the value of the `wsdl:port` element's `name` attribute.
Abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is `false`. Setting this to `true` instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.
createdFromAPI	Specifies that the user created that bean using Artix APIs like `Service.getPort()`. The default is `false`. Setting this to `true` does the following: 1. Changes the internal name of the bean by appending `.jaxws-client` to its id 3. Makes the bean abstract

In addition to the attributes listed in Table 5 it might be necessary to use multiple `xmlns:shortName` attributes to declare the

namespaces used by the `endpointName` and the `serviceName` attributes.

Adding functionality

To add functionality to your consumer or to perform advanced configuration, you must add child elements to the configuration. Child elements allow you to do the following:

- Change the endpoint's logging behavior

- Add interceptors to the endpoint's messaging chain

- Enable WS-Addressing features

- Enable reliable messaging

Table 6 describes the child elements you can use to configure JAX-WS consumer.

Table 6. Elements For Configuring a Consumer Endpoint

Element	Description
`jaxws:binding`	Specifies a bean configuring the message binding used by the endpoint. Message bindings are configured using implementations of the `org.apache.cxf.binding.BindingFactory` interface. The SOAP binding is configured using the `soap:soapBinding` bean.
`jaxws:dataBinding`	Specifies the class implementing the data binding used by the endpoint. You specify this using an embedded bean definition. The class implementing the JAXB data binding is `org.apache.cxf.jaxb.JAXBDataBinding`.
`jaxws:features`	Specifies a list of beans that configure advanced features of Artix. You can provide either a list of bean references or a list of embedded beans.
`jaxws:handlers`	Specifies a list of JAX-WS Handler implementations for processing messages. For more information in JAX-WS Handler implementations see *Writing Handlers* in ***Developing Artix Applications with JAX-WS and JAX-RS***.
`jaxws:inInterceptors`	Specifies a list of interceptors that process inbound responses.
`jaxws:inFaultInterceptors`	Specifies a list of interceptors that process inbound fault messages.

Element	Description
`jaxws:outInterceptors`	Specifies a list of interceptors that process outbound requests.
`jaxws:outFaultInterceptors`	Specifies a list of interceptors that process outbound fault messages.
`jaxws:properties`	Specifies a map of properties that are passed to the endpoint.
`jaxws:conduitSelector`	Specifies an `org.apache.cxf.endpoint.ConduitSelector` implementation for the client to use. A `ConduitSelector` implementation will override the default process used to select the `Conduit` object that is used to process outbound requests.

Example

Example 7 shows a simple consumer configuration.

Example 7. Simple Consumer Configuration

```
<beans ... xmlns:jaxws="http://cxf.apache.org/jaxws"
 ...
 schemaLocation="...
   http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
   ...">
 <jaxws:client id="bookClient"
               serviceClass="org.apache.cxf.demo.BookClientImpl"
               address="http://localhost:8080/books"/>
 ...
</beans>
```

Artix for Java Logging

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

Overview of Artix for Java Logging

Artix for Java 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

Artix comes with a default `logging.properties` file, which is located in your *InstallDir*/etc directory. This file configures both the output destination for the log messages and the message level that is published. The default configuration sets the loggers to print message flagged with the WARNING level to the console. You can either use the default file without changing any of the configuration settings or you can change the configuration settings to suit your specific application.

Logging feature

Artix includes a logging feature that can be plugged into your client or your service to enable logging. Example 8 shows the configuration to enable the logging feature.

Example 8. Configuration for Enabling Logging

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

For more information, see Logging Message Content.

Where to begin?

To run a simple example of logging follow the instructions outlined in a Simple Example of Using Logging.

For more information on how logging works in Artix, read this 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 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`,
https://docs.oracle.com/javase/6/docs/api/java/util/logging/package-summary.html.

Simple Example of Using Logging

Changing the log levels and output destination

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

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

Platform	Command
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 `InstallDir/etc` directory. It configures the Artix loggers to print `WARNING` level log messages to the console. As a result, you see very little printed to the console.

2. Stop the 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.

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 server using the following command:

Platform	Command
Windows	`start java -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\etc\mylogging.properties demo.hw.server.Server`
UNIX	`Java -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/mylogging.properties demo.hw.server.Server &`

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

Default logging.propertiesFile

The default logging configuration file, `logging.properties`, is located in the *InstallDir/etc* directory. It configures the Artix loggers to print `WARNING` level messages to the console. If this level of logging is suitable for 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. For example, you can change whether 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.

> **NOTE:** 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: https://docs.oracle.com/javase/6/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 7 shows the handlers that are configured in the default `logging.properties` file.

Table 7.   Java.util.logging Handler Classes

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

> **IMPORTANT:** 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 environment. For details on setting the Artix environment, see Using the Artix Environment Script.

Configuring the console handler

Example 9 on page 52 shows the code for configuring the console logger.

Example 9.   Configuring the Console Handler

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

The console handler also supports the configuration properties shown in Example 10.

Example 10. Console Handler Properties

```
java.util.logging.ConsoleHandler.level = WARNING ❶
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter ❷
```

The configuration properties shown in Example 10 on page 52 can be explained as follows:

❶   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 Logging Levels). The default setting is `WARNING`.

❷   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 the file handler

Example 11 shows code that configures the file handler.

Example 11. Configuring the File Handler

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

The file handler also supports the configuration properties shown in Example 12.

Example 12. File Handler Configuration Properties

```
java.util.logging.FileHandler.pattern = %h/java%u.log ❶
java.util.logging.FileHandler.limit = 50000 ❷
java.util.logging.FileHandler.count = 1 ❸
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter ❹
```

The configuration properties shown in Example 12 can be explained as follows:

❶   Specifies the location and pattern of the output file. The default setting is your home directory.

❷   Specifies, in bytes, the maximum amount that the logger writes to any one file. The default setting is `50000`. If you set it to zero, there is no limit on the amount that the logger writes to any one file.

❸   Specifies how many output files to cycle through. The default setting is `1`.

❹ 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

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 shown in Example 13.

Example 13. Configuring Both Console Logging and File Logging

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

Configuring Logging Levels

Logging levels

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

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

Configuring the global logging level

To configure the types of event that are logged across all loggers, configure the global logging level as shown in Example 14.

Example 14. Configuring Global Logging Levels

```
.level= WARNING
```

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 line of code shown in Example 15 configures logging at a SEVERE level on classes in the `com.xyz.foo` package.

Example 15. Configuring Logging at the Package Level

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

Enabling Logging at the Command Line

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 either 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 add the flag shown in Example 16 when starting the application.

Example 16. Flag to Start Logging on the Command Line

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

Logging for Subsystems and Services

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

Artix logging subsystems

Table 8 shows a list of available Artix logging subsystems.

Table 8. Artix Logging Subsystems

Subsystem	Description
`org.apache.cxf.aegis`	Aegis binding
`org.apache.cxf.binding.coloc`	colocated binding
`org.apache.cxf.binding.http`	HTTP binding
`org.apache.cxf.binding.object`	Java Object binding
`org.apache.cxf.binding.soap`	SOAP binding
`org.apache.cxf.binding.xml`	XML binding
`org.apache.cxf.bus`	Artix bus
`org.apache.cxf.configuration`	configuration framework
`org.apache.cxf.endpoint`	server and client endpoints
`org.apache.cxf.interceptor`	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.jca`	JCA container integration classes
`org.apache.cxf.js`	JavaScript front-end

Subsystem	Description
`org.apache.cxf.transport.http`	HTTP transport
`org.apache.cxf.transport.https`	secure version of HTTP transport, using HTTPS
`org.apache.cxf.transport.jms`	JMS transport
`org.apache.cxf.transport.local`	transport implementation using local file system
`org.apache.cxf.transport.servlet`	HTTP transport and servlet implementation for loading JAX-WS endpoints into a servlet container
`org.apache.cxf.ws.addressing`	WS-Addressing implementation
`org.apache.cxf.ws.policy`	WS-Policy implementation
`org.apache.cxf.ws.rm`	WS-ReliableMessaging (WS-RM) implementation
`org.apache.cxf.ws.security.wss4j`	WSS4J security implementation

Example

The WS-Addressing sample is contained in the `InstallDir/samples/cxf/ws_addressing` directory. Logging is configured in the `logging.properties` file located in that directory. The relevant lines of configuration are shown in Example 17.

Example 17. Configuring Logging for WS-Addressing

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

The configuration in Example 17 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 `InstallDir/samples/ws_addressing` directory.

Logging Message Content

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, and vice versa, complete the following steps:

1. Add the logging feature to your endpoint's configuration.

2. Add the logging feature to your consumer's configuration.

3. Configure the logging system log INFO level messages.

Adding the logging feature to an endpoint

Add the logging feature your endpoint's configuration as shown in Example 18.

Example 18. Adding Logging to Endpoint Configuration

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

The example XML shown in Example 18 enables the logging of SOAP messages.

Adding the logging feature to a consumer

Add the logging feature your client's configuration as shown in Example 19.

Example 19. Adding Logging to Client Configuration

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

The example XML shown in Example 19 enables the logging of SOAP messages.

Set logging to log INFO level messages

Ensure that the `logging.properties` file associated with your service is configured to log `INFO` level messages, as shown in Example 20.

Example 20. Setting the Logging Level to INFO

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

Logging SOAP messages

To see the logging of SOAP messages modify the wsdl_first sample application located in the `InstallDir/samples/wsdl_first` directory, as follows:

1. Add the `jaxws:features` element shown in Example 21 to the `cxf.xml` configuration file in the wsdl_first sample's directory:

Example 21. Endpoint Configuration for Logging SOAP Messages

```
<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 sample uses the default `logging.properties` file, which is located in the `InstallDir/etc` directory. Make a copy of this file and name it `mylogging.properties`.

3. In the `mylogging.properties` file, change the logging levels to `INFO` by editing the `.level` and the `java.util.logging.ConsoleHandler.level` configuration properties as follows:

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

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

Platform	Command
Windows	`start java -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\etc\ mylogging.properties demo.hw.server.Server`
UNIX	`Java -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/ mylogging.properties demo.hw.server.Server &`

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

Platform	Command
Windows	`Java -Djava.util.logging.config.file=%ARTIX_JAVA_HOME%\etc\ mylogging.properties demo.hw.client.Client .\wsdl\hello_world.wsdl`
UNIX	`Java -Djava.util.logging.config.file=$ARTIX_JAVA_HOME/etc/ mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl`

The SOAP messages are logged to the console.

Deploying to the Spring Container

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

Introduction

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 service endpoints in the Spring container.

Figure 1 shows how to access a deployed Artix endpoint in the Spring container.

Figure 1. Artix Endpoint Deployed in a Spring Container
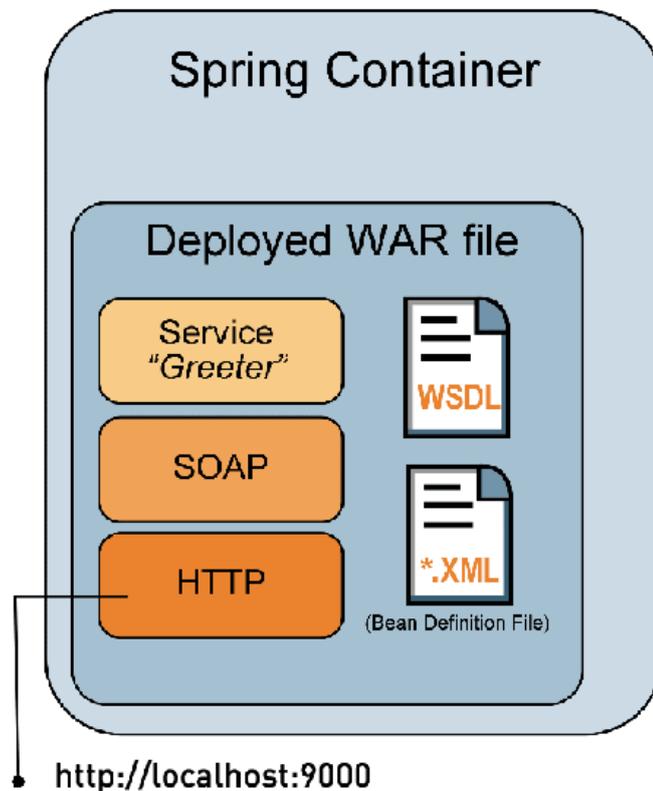
You deploy a Web Archive (WAR) file to the Spring container. The WAR file contains all of the files that the Spring container needs to run your application. These include the WSDL file that defines your service, the code that you generated from the WSDL file, including the implementation file, and any libraries that your application requires. It also includes an Artix runtime Spring-based XML configuration file to configure your application.

The Spring container loads each WAR file using a unique class loader. The class loader incorporates a firewall class loader that ensures that any classes contained in the WAR are loaded before classes in the parent class loader are loaded.

Sample XML

The example XML used in this chapter is taken from the Spring container sample application located in:

```
InstallDir/samples/spring_container
```

Running the Spring Container

This section explains how to run the Spring container using the **spring_container** command.

Using the spring_container command

The **spring_container** command is located in the *InstallDir*/bin directory, and has the following syntax:

```
spring_container [-config spring-config-url] [-wsdl
container-wsdl-url] [-h] [-verbose] [[start] | [stop]]
```

Table 9. Spring Container Command Options

Option	Description
-config spring-config-url	Specifies the URL or file location of the Spring container configuration file, which is used to launch the Spring container. This flag is optional.
	By default, the Spring container uses the spring_container.xml file, which is located in the *InstallDir*/containers/spring_container/etc directory.
	You should only use the -config flag if you are specifying a different configuration file. For example, see Running Multiple Containers on Same Host.

Option	Description
`-wsdl container- wsdl-url`	Specifies the URL or file location of the Spring container WSDL file. This flag is optional. By default, the Spring container uses the `container.wsdl` file located in the `InstallDir/containers/spring_container/etc/wsdl` directory. You should only use the `-wsdl` flag if you are specifying a different Spring container WSDL file. For example, see Running Multiple Containers on Same Host.
`-h`	Prints usage summary and exits. This flag is optional.
`-v`	Specifies verbose mode. This flag is optional.
`<start|stop>`	Starts and stops the Spring container. These flags are required to start and stop the Spring container respectively.

Starting the Spring container

To start the Spring container, run the following command from the `InstallDir/bin` directory:

```
spring_container start
```

If you wish to start more than one container on a single host, see Running Multiple Containers on Same Host.

Stopping the Spring container

To stop the Spring container, run the following command from the `InstallDir/bin` directory:

```
spring_container stop
```

If you are running more than one container on the same host, see Running Multiple Containers on Same Host.

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 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 requires. See Building a WAR file.

3. Deploy the WAR file in one of the following ways:

    - Copy the WAR file to the Spring container repository. See Deploying the WAR file to the Spring repository.

    - Use the JMX console. See Managing the Container using the JMX Console.

    - Use the Web service interface. See Managing the Container using the Web Service Interface.

Configuring your application

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

Example 22 shows the `spring.xml` configuration file used in the Spring container sample application. You can use any name for your configuration file, however, it must end with a `.xml` extension. This example file is taken from the `InstallDir/samples/spring_container` sample application.

Example 22. Configuration File—spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ❶
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    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">
    <jaxws:endpoint id="SoapEndpoint" ❷
        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"/>
    <bean id="SOAPServiceImpl" class="demo.hw.server.GreeterImpl"/> ❸
</beans>
```

The code shown in Example 22 can be explained as follows:

❶ The Spring `beans` element is required at the beginning of every Artix configuration file. It is the only Spring element that you must be familiar with.

❷ Configures a `jaxws:endpoint` element that defines a service and its corresponding endpoints. (For more information on configuring an Artix `jaxws:endpoint` element, see Using the jaxws:endpoint Element.)

IMPORTANT: The location of the WSDL file specified in the `wsdlLocation` is relative to the WAR's `WEB_INF/wsdl` folder.

❸ Identifies the class that implements the service.

Building a WAR file

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

- `META-INF/spring` should include your configuration file. The configuration file must have a `.xml` extension.

- `WEB-INF/classes` should include your Web service implementation class, and any other classes (including the class hierarchy) generated by the wsdl2java utility. For details, see *wsdl2java* in ***Artix Java Runtime Command Reference***.

- `WEB-INF/wsdl` should include the WSDL file that defines the service that you are deploying.

- `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:

   ```
   InstallDir/bin/spring_container  start
   ```

2. Copy the WAR file to the Spring container repository. The default location for the repository is
   `InstallDir/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.

Using Ant to build a WAR file and deploy to the Spring container

You can use the Apache **ant** utility to build the Artix sample applications. This includes building the WAR files and deploying them to the Spring container. If you want to use the **ant** utility to build your application war files for deployment to the Spring container, see the 'create.spring.war' target in `common_build.xml` file located in the *InstallDir*/samples directory.

For more information about the **ant** utility, see http://ant.apache.org/.

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 23 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 23 for more detail.

Managing the Container using the JMX Console

You can use the JMX console to deploy and manage applications in the Spring container. The JMX console enables you to 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 *InstallDir*/bin directory:

Platform	Command
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).

The `SpringContainer` MBean is deployed as part of the Spring container. It provides access to the management interface for the Spring Container and can be used to deploy, stop, start, remove and list applications. I can also get information on an application's state.

Figure 2. JMX Console—SpringContainer MBean
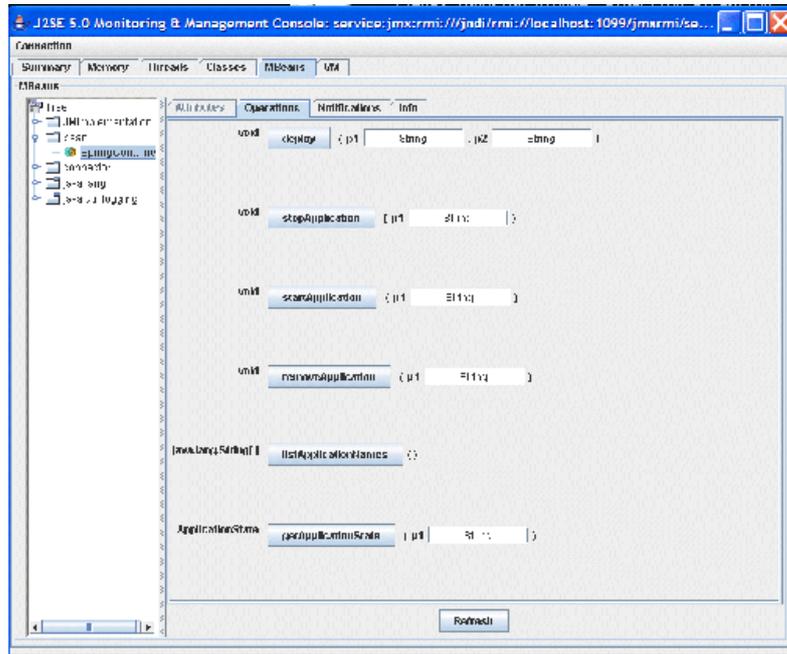
The operations and their parameters are described in Table 10.

Table 10. 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 and puts the copy 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.

Operation	Description	Parameters
`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 whether an application 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

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 *InstallDir*`/containers/spring_container/etc/wsdl` directory of your installation.

Client tool

Artix does not currently include a client tool for the Web service interface. However, you can write one if you are familiar with Web service development. Please see the `container.wsdl` file and *Developing Artix Applications with JAX-WS and JAX-RS* for more details.

Changing the port the Web service interface listens on

To change the port that the Web service interface listens on, you must change the port number of the address property in the `spring_container.xml` file, as shown:

```
<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.

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

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

InstallDir/containers/spring_container/etc

spring_container.xml

The contents of the Spring container configuration file are shown in Example 23.

Example 23. spring_container.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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 -->
 <container:container id="container"
     containerRepository="${ARTIX_JAVA_HOME}/containers/spring_container/repository"
     scanInterval="5000"/> ❶

 <!-- Web Service Container Management -->
 <jaxws:endpoint id="ContainerService" ❷
                 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  -->
 <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"> ❸

   <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.InterfaceBasedMBean InfoAssembler">
     <property name="interfaceMappings">
       <props>
         <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://local
host:1099/jmxrmi/server"/>
   </bean>

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

The XML shown in Example 23 does the following:

❶ 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
   `InstallDir/containers/spring_container/repository`. You
   can change the repository location by changing the value
   of the `containerRepository` attribute.

   The `scanInterval` attribute 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.

❷ 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.

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

For more information, see the JMX chapter of the Spring 3.0.x reference document available at http://static.springframework.org/spring/docs/3.0.x/reference/jmx.html.

Running Multiple Containers on Same Host

You might want to run more than one instance of a Spring container on a single host. This allows you to load balance between multiple containers and also allows you to separate applications. Setting up multiple Spring containers to run on a single host requires you to modify each container's configuration so that there are no resource clashes.

Procedure

If you want to run more than one Spring container on the same host, you must do the following:

1. Make a copy of the container.wsdl file, which is located in the *InstallDir*/containers/spring_container/etc/wsdl directory.

2. In your new copy, my_container.wsdl, change the port on which the Web service interface listens from 2222 to another port by changing the address property as shown below:

```
<service name="ContainerService">
  <port name="ContainerServicePort" binding="tns:ContainerServiceBinding">
  <soap:address location="http://localhost:2222/AdminContext/AdminPort"/>
 </port>
</service>
```

3. Make a copy of the spring_container.xml file, which is located in the *InstallDir*/containers/spring_container/etc directory.

4. Make the following changes to your new copy, *my_spring_container*.xml:

- Container repository location—change the container's containerRepository property to point to a new repository. For example, you change:

```
<container:container id="container"
 containerRepository="${ARTIX_JAVA_HOME}/containers/spring_container/repository"
 scanInterval="5000"/>
```

To:

```
<container:container id="container"
 containerRepository="MyNewContainerRepository/spring_container/repository"
 scanInterval="5000"/>
```

- Change the port on which the Web service interface listens by changing the address property as follows:

```
<jaxws:endpoint id="ContainerService" implementor="#ContainerServiceImpl"
        address=" http://localhost:2222/AdminContext/AdminPort">0
```

- Change the JMX port from 1099 to a new port as show in the following line:

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

- Change the RMI registry port from 1099 to a new port as shown in the following snippet:

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

5. Make a copy of the JMX console launch script, jmx_console_start.bat, which is located in the *InstallDir*/bin directory.

6. Change the following line in the copy of the JMX console launch script to point to the JMX port that is specified above:

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

7. Start the new container by passing the URL, or file location of its configuration file, my_spring_container.xml, to the start_container script as follows:

```
InstallDir/bin/spring_container -config my_spring_contain  er.xml start
```

8. To view the new container using the JMX console, run the JMX console launch script created in steps 5 and 6.

9. Stop the new container by passing the URL or file location of its WSDL file, my_container.wsdl, to the **spring_container** command.

   For example, if the my_container.wsdl file has been saved to the *InstallDir*/containers/spring_container/wsdl directory, run the following command:

```
InstallDir/bin/spring_container  -wsdl
..\containers\spring_container\wsdl\my_container.wsdl  stop
```

Deploying to a Servlet Container

Artix endpoints can be deployed into any servlet container. Artix provides a standard servlet adapter that works for most service providers. It is also possible to deploy Artix endpoints using a Spring context or by creating a custom servlet to instantiate the Artix endpoint.

Introduction

Servlet containers are a common platform for running Web services. The Artix runtime's light weight and plugability make it easy to deploy endpoints into a servlet container. There are several ways to deploy endpoints into a servlet container:

- an Artix provided servlet adapter class

- a custom servlet

- the Spring servlet context listener

- the Artix JCA connector

NOTE: Not all servlet containers support JCA connectors.

Deploying service providers

The preferred way to deploy a service provider into a servlet container is to use the CXF servlet. The CXF servlet only requires a few additional pieces of configuration to deploy a service provider into the servlet container. Much of the additional information is either canned information required deploy the servlet or Artix configuration for the endpoint.

It is also possible to deploy a service provider using any of the other methods.

Deploying service consumers

Service consumers cannot be deployed using the CXF servlet. They can be deployed using either a custom servlet that creates the required proxies or using the Artix JCA adapter.

Configuring the Servlet Container

Before you can deploy an Artix endpoint to your servlet container you must make the Artix runtime libraries available to the container. There are two ways to accomplish this:

1. Add the required libraries to the container's shared library folder

   This approach has the advantage of keeping individual WAR files small. It also ensures that all of the Artix servlets are using the same version of the libraries.

2. Add the required libraries to each application's WAR file

   This approach has the advantage of flexibility. Each WAR can contain the versions of the libraries it requires.

Required libraries

Artix endpoints require all of the JAR files in the `InstallDir`/lib directory except the following:

- `artix.jar`

- `it-soa-jaxwsgenerator*.jar`

- `it-soa-router.jar`

- `it-soa-transport-mq*`

- `servlet-api*.jar`

- `geronimo-servlet_*.jar`

- `jetty-*.jar`

Automating servlet container configuration

The Artix samples directory, `InstallDir`/samples, includes a `common_build.xml` file that contains utilities that automates the configuration of the servlet environment.

One utility is the **copy-war-libs** Ant target. It copies the required libraries to the folder specified in the war-lib. For example, to install the required libraries into a Tomcat 6 installation enter **ant copy-war-libs -Dwar-lib=`CATALINA_HOME`\lib**.

The other utility is the **cxfwar** macro. The macro is used to build the WAR files for all of the Artix samples. Its default result is to make a WAR containing all of the required libraries. This behavior can be changed by setting the without.libs property to `true`.

Using the CXF Servlet

Artix provides a standard servlet, the CXF servlet, which acts as an adapter for the Web service endpoints. The CXF servlet is the easiest method for deploying Web services into a servlet container.

Figure 3 shows the main components of an Artix endpoint deployed using the CXF servlet.

Figure 3. Artix Endpoint Deployed in a Servlet Container
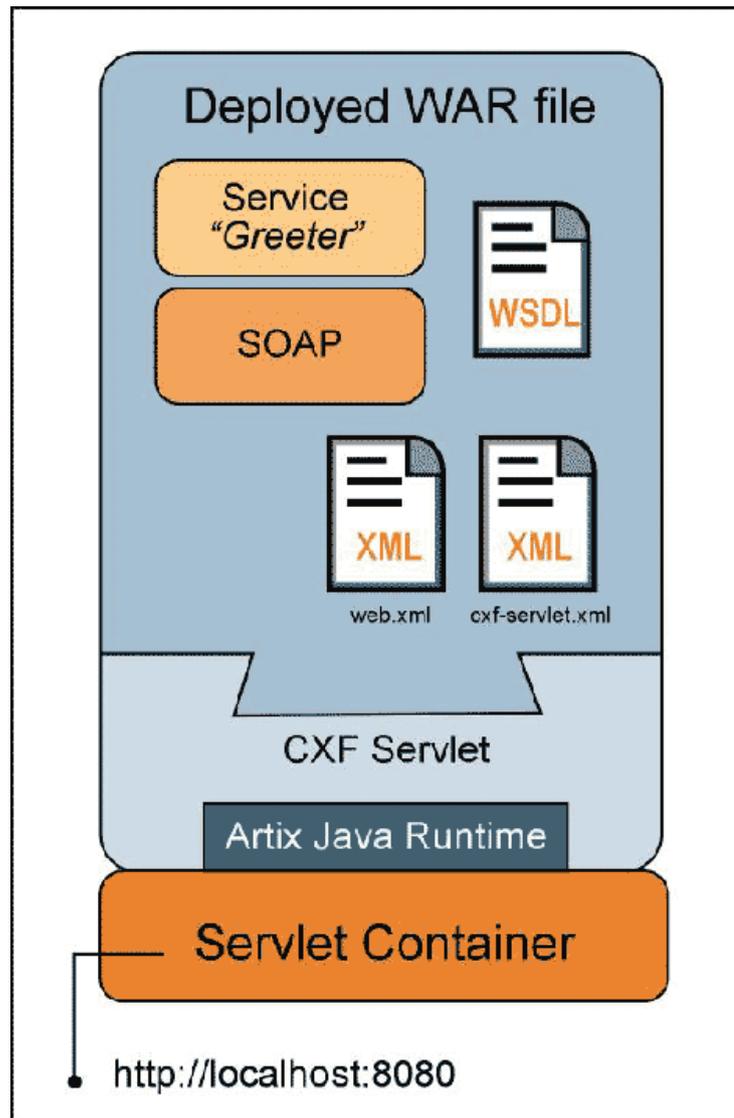
- Deployed WAR file — Service providers are deployed to the servlet container in a Web Archive (WAR) file. The deployed WAR file contains:

- the compiled code for the service provider being deployed

- the WSDL file defining the service

- the Artix configuration file

  This file, called `cxf-servlet.xml`, is standard Artix configuration file that defines all of the endpoints contained in the WAR.

- the Web application deployment descriptor

  All Artix Web applications using the standard CXF servlet need to load the `org.apache.cxf.transport.servlet.CXFServlet` class.

- CXF servlet — The CXF servlet is a standard servlet provided by Artix. It acts as an adapter for Web service endpoints and is part of the Artix runtime. The CXF servlet is implemented by the `org.apache.cxf.transport.servlet.CXFServlet` class.

Deployment steps

To deploy an Artix endpoint to a servlet container you must:

- Build a WAR that contains your application and all the required support files.

- Deploy the WAR file to your servlet container.

Building a WAR

To deploy your application to a servlet container, you must build a WAR file.

The WAR file's `WEB-INF` folder should include the following:

- `cxf-servlet.xml` — an Artix configuration file specifying 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 Servlet configuration file for more information.

- `web.xml` — a standard web application file that instructs the servlet container to load the `org.apache.cxf.transport.servlet.CXFServlet` class.

  A reference version of this file is contained in your `InstallDir/etc` directory. You can use this reference copy without making changes to it.

- `classes` — a folder including your Web service implementation class and any other classes required to support the implementation.

- wsdl — a folder including the WSDL file that defines the service you are deploying.

- lib — a folder including any JARs required by your application.

Servlet configuration file

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

Example 24 on page 88 shows a simple `cxf-servlet.xml` file.

Example 24. CXF Servlet Configuration File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ❶
       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.springframe work.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">
    <jaxws:endpoint ❷
        id="hello_world"
        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 24 is explained as follows:

❶ The Spring `beans` element is required at the beginning of every Artix configuration file. It is the only Spring element that you need to be familiar with.

❷ The `jaxws:endpoint` element defines a service provider endpoint. The `jaxws:endpoint` element has the following attributes:

- `id` — Sets the endpoint id.

- `implementor` — Specifies the class implementing the service. This class needs to be included in the WAR's `WEB-INF/classes` folder.

- `wsdlLocation` — Specifies the WSDL file that contains the service definition. The WSDL file location is relative to the WAR's `WEB-INF/wsdl` folder.

- `address` — Specifies the address of the endpoint as defined in the service's WSDL file that defines service that is being deployed.

- `jaxws:features` — Defines features that can be added to your endpoint.

For more information on configuring a `jaxws:endpoint` element, see Using the jaxws:endpoint Element.

Web application configuration

You must include a `web.xml` deployment descriptor file that instructs the servlet container to load the CXF servlet. Example 25 shows a `web.xml` file. It is not necessary to change this file. A reference copy is located in the `InstallDir/etc` directory.

Example 25. A web.xml Deployment Descriptor File

```xml
<?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>
```

Deploying a 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, you copy it to the Tomcat `CATALINA_HOME/server/webapp` directory.

Using a Custom Servlet

In some cases, you might want to write a custom servlet that deploys Artix enabled endpoints. A common reason is to deploy Artix client applications into a servlet container. The CXF servlet does not support deploying pure client applications.

Procedure

The procedure for using a custom servlet is similar to the one for using the default CXF servlet:

1. Implement a servlet that instantiates an Artix enabled endpoint.

2. Package your servlet in a WAR that contains the Artix libraries and the configuration needed for your application.

3. Deploy the WAR to your servlet container.

Differences from using the default servlet

There are a few important differences between using the CXF servlet and using a custom servlet:

- The configuration file is not called `cxf-servlet.xml`.

  The default behavior is similar to that of a regular Artix application. It looks for its configuration in a file called `WEB-INF/classes/cxf.xml`. If you want to locate the configuration in a different file, you can programmatically configure the servlet to load the configuration file.

- Any paths in the configuration file are relative to the servlet's `WEB-INF/classes` folder.

Implementing the servlet

Implementing the servlet is easy. You simply add logic to the servlet's constructor to instantiate the Artix endpoint. Example 26 shows an example of instantiating a consumer endpoint in a servlet.

Example 26. Instantiating a Consumer Endpoint in a Servlet

```
public class HelloWorldServlet extends HttpServlet
{
  private static Greeter port;

  public HelloWorldServlet()
  {
    URL wsdlURL = getClass().getResource("/hello_world.wsdl");

     port = new SOAPService(wsdlURL,
       new QName("http://apache.org/hello_world_soap_http",
       "SOAPService")).getSoapPort();
  }

  ...
  }
```

If you choose not to use the default location for the configuration file, then you must add code for loading the configuration file. To load the configuration from a custom location do the following:

1. Use the `ServletContext` to resolve the file location into a URL.

2. Create a new bus for the application using the resolved URL.

3. Set the application's default bus to the newly created bus.

shows an example of loading the configuration from the `WEB-INF/client.xml` file.

Example   27.   Loading Configuration from a Custom Location

```
public class HelloWorldServlet extends HttpServlet
{
  public init(ServletConfig cfg)
  {
    URL configUrl=cfg.getServletContext().getResource("WEB-
INF/client.xml");
    Bus bus = new SpringBusFactory().createBus(url);
    BusFactory.setDefaultBus(bus);
  }

  ...
}
```

Depending on what other features you want to use, you might need to add additional code to your servlet. For example, if you want to use WS-Security in a consumer you must add code to your servlet to load the credentials and add them to your requests.

Building the WAR file

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

- The `WEB-INF` folder should include a `web.xml` file which instructs the servlet container to load the custom servlet.

- The `WEB-INF/classes` folder should include the following:

  - The implementation class and any other classes (including the class hierarchy) generated by the wsdl2java utility

  - The default `cxf.xml` configuration file

  - Other resource files that are referenced by the configuration.

- The `WEB-INF/wsdl` folder should include the WSDL file that defines the service being deployed.

- The `WEB-INF/lib` folder should include any JARs required by the application.

Using the Spring Context Listener

An alternative approach to instantiating endpoints inside a servlet container is to use the Spring context listener. The Spring context listener provides more flexibility in terms of how an application is wired together. It uses the application's Spring configuration to determine what object to instantiate and loads the objects into the application context used by the servlet container.

The added flexibility adds complexity. The application developer must know exactly what application components need to loaded. They also must know what Artix components need to be loaded. If any component is missing, the application will not load properly and the desired endpoints will not be created.

Procedure

The following steps are involved in building and packaging a Web application that uses the Spring context listener:

1. Develop the application's business logic.

   Only the service implementation needs to be developed service provider endpoints.

   The business logic for service consumers should be encapsulated in a Java class and not as part of the `main()` method.

2. Update the application's `web.xml` file to load the Spring context listener and the application's Spring configuration.

3. Create a Spring configuration file that explicitly loads all of the application's components and all of the required Artix components.

4. Package the application into a WAR file for deployment.

Configuring the Web application

The servlet container looks in the `WEB-INF/web.xml` file to determine what classes are needed to activate the Web application. When deploying an Artix based application using the Spring context listener, the servlet container needs to load the `org.springframework.web.context.ContextLoaderListener` class. This is specified using the `listener` element and its `listener-class` child.

The `org.springframework.web.context.ContextLoaderListener` class uses a context parameter called *contextConfigLocation* to determine the location of the Spring configuration file. The context parameter is configured using the `context-parameter` element. The `context-param` element has two children that specify parameters and their values. The `param-name` element

specifies the parameter's name. The `param-value` element specifies the parameter's value.

Example 28shows a `web.xml` file that configures the servlet container to load the Spring listener and a Spring configuration file.

Example 28. Web Application Configuration for Loading the Spring Context Listener

```
<!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>
  <context-param> ❶
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/beans.xml</param-value>
  </context-param>

  <listener> ❷
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>

  </listener>
  ...
</web-app>
```

The XML in Example 28 does the following:

❶ Specifies that the Spring context listener will load the application's Spring configuration from `WEB-INF/beans.xml`.

❷ Specifies that the servlet container should load the Spring context listener.

Creating the Spring configuration

The Spring configuration file for an application using the Spring context listener is similar to a standard Artix configuration file. It uses all of the same endpoint configuration elements described in Configuring Artix Endpoints. It can also contain standard Spring beans.

The difference between a typical Artix configuration file and a configuration file for using the Spring context listener is that the Spring context listener configuration must import the configuration for all of the Artix runtime components used by the endpoint's exposed by the application. These components are imported into the configuration as resources using an `import` element for each component's configuration.

Example 29 shows the configuration for a simple consumer endpoint being deployed using the Spring context listener.

Example 29. Configuration for a Consumer Deployed into a Servlet Container Using the Spring Context Listener

```
<beans ... >

  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-jaxws.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-http-
  binding.xml" />
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

  <jaxws:client id="funguy"
                address="http://localhost:9000/funguyTool"
                serviceClass="org.laughs.funGuyImpl" />

</beans>
```

The `import` elements at the beginning of Example 29 import the required Artix component configuration. The required Artix component configuration files depends on the features being used by the endpoints. At a minimum, an application in a servlet container will need the components shown in Example 29.

TIP: Importing the `cxf-all.xml` configuration file will automatically import all of the Artix components.

Building the WAR

To deploy your application to a servlet container, you must build a WAR file. The `WEB-INF` folder should include the following:

- `beans.xml` — the Spring configuration file configuring the application's beans.

- `web.xml` — the web application file that instructs the servlet container to load the Spring context listener.

- `classes` — a folder including the Web service implementation class and any other classes required to support the implementation.

- `wsdl` — a folder including the WSDL file that defines the service being deployed.

- `lib` — a folder including any JARs required by the application.

Deploying WS-Addressing

Artix supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Artix runtime environment.

Introduction to WS-Addressing

WS-Addressing is a specification that allows services to communicate addressing information in a transport neutral way. It consists of two parts:

- A structure for communicating a reference to a Web service endpoint

- A set of Message Addressing Properties (MAP) that associate addressing information with a particular message

Supported specifications

Artix supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification.

Further information

For detailed information on WS-Addressing, see the 2004/08 submission at http://www.w3.org/Submission/ws-addressing/.

WS-Addressing Interceptors

In Artix, WS-Addressing functionality is implemented as interceptors. The Artix runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the WS-Addressing interceptors are added to the application's interceptor chain, any WS-Addressing information included with a message is processed.

The WS-Addressing implementation consists of two interceptors, as described in Table 11.

Table 11. WS-Addressing Interceptors

Interceptor	Description
`org.apache.cxf.ws.addressing.MAPAggregator`	A logical interceptor responsible for aggregating the Message Addressing Properties (MAPs) for outgoing messages.
`org.apache.cxf.ws.addressing.soap.MAPCodec`	A protocol-specific interceptor responsible for encoding and decoding the Message Addressing Properties (MAPs) as SOAP headers.

Enabling WS-Addressing

To enable WS-Addressing the WS-Addressing interceptors must be added to the inbound and outbound interceptor chains. This is done in one of the following ways:

- Artix Features

- RMAssertion and WS-Policy Framework

- Using Policy Assertion in a WS-Addressing Feature

Adding WS-Addressing as a Feature

WS-Addressing can be enabled by adding the WS-Addressing feature to the client and the server configuration as shown in Example 30 and Example 31 respectively.

Example 30. client.xml—Adding WS-Addressing Feature to 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:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-
       beans.xsd">

    <jaxws:client ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:client>
```

Example 31. server.xml—Adding WS-Addressing Feature to Server Configuration

```xml
<?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:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.spring
framework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Configuring WS-Addressing Attributes

The Artix WS-Addressing feature element is defined in the namespace http://cxf.apache.org/ws/addressing. It supports the two attributes described in Table 12.

Table 12. WS-Addressing Attributes

Attribute Name	Value
allowDuplicates	A boolean that determines if duplicate MessageIDs are tolerated. The default setting is true.
usingAddressingAdvisory	A boolean that indicates if the presence of the UsingAddressing element in the WSDL is advisory only; that is, its absence does not prevent the encoding of WS-Addressing headers.

Configuring WS-Addressing attributes

Configure WS-Addressing attributes by adding the attribute and the value you want to set it to the WS-Addressing feature in your server or client configuration file. For example, the following configuration extract sets the `allowDublicates` attribute to `false` on the server endpoint:

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing"
...>
    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing allowDuplicates="false"/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Using a WS-Policy assertion embedded in a feature

In Example 32 an addressing policy assertion to enable non-anonymous responses is embedded in the `policies` element.

Example 32. Using the Policies to Configure WS-Addressing

```
<?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:policy="http://cxf.apache.org/policy-config"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       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.springframe work.org/schema/beans/spring-beans.xsd">

  <jaxws:endpoint
name="{http://cxf.apache.org/greeter_control}GreeterPort" createdFromAPI=
"true">
        <jaxws:features>
          <policy:policies>
            <wsp:Policy
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
              <wsam:Addressing>
                <wsp:Policy>
                <wsam:NonAnonymousResponses/>
              </wsp:Policy>
            </wsam:Addressing>
          </wsp:Policy>
        <policy:policies>
      </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Enabling Reliable Messaging

Artix supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in Artix.

Introduction to WS-RM

WS-ReliableMessaging (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.

How WS-RM works

WS-RM ensures the reliable delivery of messages between a source and a 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
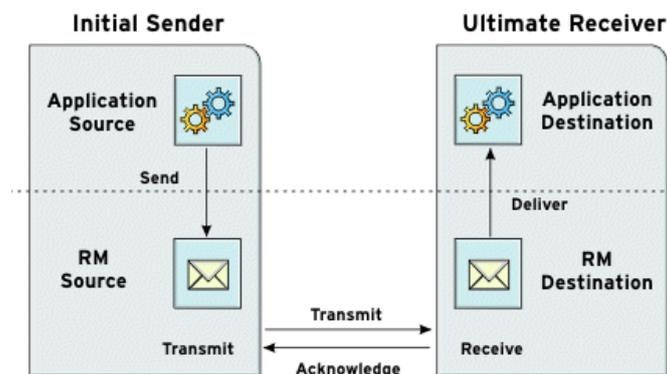
The flow of WS-RM messages can be described as follows:

1. The RM source sends a `CreateSequence` protocol message to the RM destination. This contains a reference for the endpoint that receives acknowledgements (the `wsrm:AcksTo` endpoint).

2. The RM destination sends a `CreateSequenceResponse` protocol message back to the RM source. This message 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 header 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 that 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.

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. Either the source or the destination endpoint logs an error if reliable delivery cannot 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

In Artix, WS-RM functionality is implemented as interceptors. The Artix runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the application's interceptor chain includes the WS-RM interceptors, the application can participate in reliable messaging sessions. The WS-RM interceptors handle the collection and aggregation

of the message chunks. They also handle all of the acknowledgement and retransmission logic.

Artix WS-RM Interceptors

The Artix WS-RM implementation consists of four interceptors, which are described in Table 13.

Table 13. Artix 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.

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, when intercepting the first application message on the outbound interceptor chain, the `RMOutInterceptor` sends a `CreateSequence` request and waits to process the original application message until 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. It is not necessary 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.

Configuring WS-RM Attributes

You 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. To enforce the use of a separate sequence per application message configure 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.

Enabling WS-RM

To enable reliable messaging, the WS-RM interceptors must be added to the interceptor chains for both inbound and outbound messages and faults.

Because the WS-RM interceptors use WS-Addressing, 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

- Implicitly, using WS-Policy assertions, which cause the Artix runtime to transparently add the interceptors on your behalf.

Spring beans—explicitly adding interceptors

To enable WS-RM add the WS-RM and WS-Addressing interceptors to the Artix bus, or to a consumer or service endpoint using Spring bean configuration. This is the approach taken in the WS-RM sample that is found in the `InstallDir`/`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 33).

Example 33. Enabling WS-RM Using Spring Beans

```
<?xml version="1.0" encoding="UTF-8"?>
❶<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">
❷   <bean id="mapAggregator"
class="org.apache.cxf.ws.addressing.MAPAggregator"/>
    <bean id="mapCodec" class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
❸   <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">
❹          <property name="inInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
❺          <property name="inFaultInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalIn"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
❻          <property name="outInterceptors">
            <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
❼          <property name="outFaultInterceptors">
            <list>
                <ref bean="mapAggregator">
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
    </bean>
</beans>
```

The code shown in Example 33 can be explained as follows:

❶ An Artix configuration file is 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.

❷ Configures each of the WS-Addressing interceptors—MAPAggregator and MAPCodec. For more information on WS-Addressing, see Deploying WS-Addressing.

❸ Configures each of the WS-RM interceptors—RMOutInterceptor, RMInInterceptor, and RMSoapInterceptor.

❹ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.

❺ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.

❻ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.

❼ Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound faults.

WS-Policy framework—implicitly adding interceptors

The 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 WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint.

Example 34 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 34. 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>
```

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—to your WSDL file, as shown in Example 35.

Example 35. 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-200401-wss-
wssecurity-utility- 1.0.xsd">
    <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

You can configure WS-RM by:

- Setting Artix-specific attributes that are defined in the Artix
  WS-RM manager namespace,
  http://cxf.apache.org/ws/rm/manager.

- Setting standard WS-RM policy attributes that are defined in
  the http://schemas.xmlsoap.org/ws/2005/02/rm/policy
  namespace.

Configuring Artix-Specific WS-RM Attributes

To configure the Artix-specific attributes, use the `rmManager`
Spring bean. Add the following to your configuration file:

- The http://cxf.apache.org/ws/rm/manager namespace to your
  list of namespaces.

- An `rmManager` Spring bean for the specific attribute that your
  want to configure.

Example 36 shows a simple example.

Example 36. Configuring Artix-Specific WS-RM Attributes

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframe work.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager
 http://cxf.apache.org/schemas/configuration/wsrm-man ager.xsd">
...
<wsrm-mgr:rmManager>
<!--
  ...Your configuration goes here
-->
</wsrm-mgr:rmManager>
```

Children of the rmManager Spring bean

Table 14 shows the child elements of the rmManager Spring bean, defined in the http://cxf.apache.org/ws/rm/manager namespace.

Table 14. Children of the rmManager Spring Bean

Element	Description
RMAssertion	An element of type RMAssertion
deliveryAssurance	An element of type DeliveryAssuranceType that describes the delivery assurance that should apply
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

Example

For an example, see Maximum unacknowledged messages threshold.

Configuring Standard WS-RM Policy Attributes

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

- RMAssertion in rmManager Spring bean

- Policy within a feature

- WSDL file

- External attachment

WS-Policy RMAssertion Children

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

Table 15. Children of the WS-Policy RMAssertion Element

Name	Description
InactivityTimeout	Specifies the amount of time that must pass without receiving a message before an endpoint can consider an 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.
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 Spring bean

You can configure standard WS-RM policy attributes by adding an `RMAssertion` within an Artix `rmManager` Spring bean. This is the best 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 37 shows:

- A standard WS-RM policy attribute,
  `BaseRetransmissionInterval`, configured using an `RMAssertion`
  within an `rmManager` Spring bean.

- An Artix-specific RM attribute, `intraMessageThreshold`,
  configured in the same configuration file.

Example 37. Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...>
  <wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
    <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 38.

Example 38. 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-200401-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.springframe
work.org/schema/beans/spring-beans.xsd">
  <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
           created FromAPI="true">
    <jaxws:features>
      <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 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 WS-Policy framework—implicitly adding interceptors where 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 39 shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 39. 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>
              <wsa:Address>http://localhost:9020/SoapContext/GreeterPort
              </wsa: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

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` 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.

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.

Example 40 shows how to set the WS-RM base retransmission interval.

Example 40. Setting the WS-RM Base Retransmission Interval

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<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. An exponential backoff ratio of `2` is used by default.

shows how to set the WS-RM exponential backoff for retransmission.

Example 41. Setting the WS-RM Exponential Backoff Property

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

shows how to set the WS-RM acknowledgement interval.

Example 42. Setting the WS-RM Acknowledgement Interval

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

Example 43 on page 125 shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 43. Setting the WS-RM Maximum Unacknowledged Message Threshold

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</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.

Example 44 shows how to set the maximum length of an RM sequence.

Example 44. Setting the Maximum Length of a WS-RM Message Sequence

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
  <wsrm-mgr:reliableMessaging>
     <wsrm-mgr:sourcePolicy>
         <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
     </wsrm-mgr:sourcePolicy>
  </wsrm-mgr:reliableMessaging>
</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 only once. If a message is delivered more than once an error is 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.

Example 45 shows how to set the WS-RM message delivery
assurance.

**Example 45. Setting the WS-RM Message Delivery
Assurance Policy**

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

Configuring WS-RM Persistence

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 a RM destination crash.

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

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. To implement your own persistence
mechanism, you can implement one using this API with your
preferred DB.

IMPORTANT: WS-RM persistence is supported for oneway calls only,
and 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 you can 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 16.

Table 16. 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 shown in Example 46 enables the JDBC-based store that comes with Artix, while setting the driverClassName and url to non-default values.

Example 46. Configuring the JDBC Store for WS-RM Persistence

```
<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>
```

Enabling High Availability

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

Introduction to High Availability

Scalable and reliable 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 `/java/samples/ha` directory of your Artix installation.

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

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 47 shows a WSDL file extract that defines a service cluster of three replicas.

Example 47. Enabling HA with Static Failover—WSDL File

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

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

❹    <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 47 can be explained as follows:

❶ Defines a service, `ClusterService`, which is exposed on three ports:

- `Replica1`

- `Replica2`

- `Replica3`

❷ Defines `Replica1` to expose the `ClusterService` as a SOAP over HTTP endpoint on port `9001`.

❸ Defines `Replica2` to expose the `ClusterService` as a SOAP over HTTP endpoint on port `9002`.

❹ Defines `Replica3` to expose 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 48.

Example 48. Enabling HA with Static Failover—Client Configuration

```xml
<?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/>
       </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

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's 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 49 to your client configuration file:

**Example 49. Configuring a Random Strategy for Static
Failover**

```
<beans ...>
❶     <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>
❷                <clustering:strategy>
                    <ref bean="Random"/>
                </clustering:strategy>
            </clustering:failover>
        </jaxws:features>
    </jaxws:client>
</beans>
```

The configuration shown in Example 49 can be explained as
follows:

❶   Defines a Random bean and implementation class that
    implements the random strategy.

❷   Specifies that the random strategy be used when selecting a
    replica.

Enabling HA with Dynamic Failover

To enable HA with dynamic failover, you do the following:

- Configure your service to register with the Artix locator

- Configure your client to use locator meditated failover

- 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 50 to your server configuration file.

Example 50. Configuring Your Service to Register with the Locator

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:locatorEndpoint="http://com.iona.soa/discovery/locator/endpoint"
...>

  <!-- Configuration for Locator runtime support -->
❶    <bean id="LocatorSupport"
         class="com.iona.soa.discovery.locator.rt.cxf.LocatorSupport">
         <property name="bus" ref="cxf"/>
         <property name="contract">
             <value>http://localhost:9000/services/LocatorService</value>
         </property>
    </bean>

    <jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                  createdFromAPI="true">
❷        <jaxws:features>
            <locatorEndpoint:registerOnPublish monitorLiveness="true"
                                                heartbeatInterval="10001" />
        </jaxws:features>
    </jaxws:endpoint>

</beans>
```

The configuration shown in Example 50 is taken from the HA sample and can be explained as follows:

❶ Enables the service to use the Artix locator service.

❷ 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 51 to your client configuration file.

Example 51. Configuring your Client to Use Locator Mediated Failover

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

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

❷    <bean id="LocatorMediated"
         class="com.iona.soa.failover.locator.rt.cxf.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 51 is from the HA sample and can be explained as follows:

❶ Enables the client to use the Artix locator service to find services.

❷ 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

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 attemptsSequential 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.soa.failover.
locator.rt.cxf.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.soa.failover.
locat or.rt.cxf.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.soa.failover.
locator.rt.cxf.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.soa.failover.
locator.rt.cxf.LocatorMediatedStrategy">
     <property name="backoff" value="1.2"/>
</bean>
```

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.

Artix WSDL Publishing Service

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.

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.soa.wsdlpublish.rt.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 on page 143 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 the WSDL file, and *QueryString* is a string that requests a particular WSDL file (see Querying the WSDL Publishing Service). 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
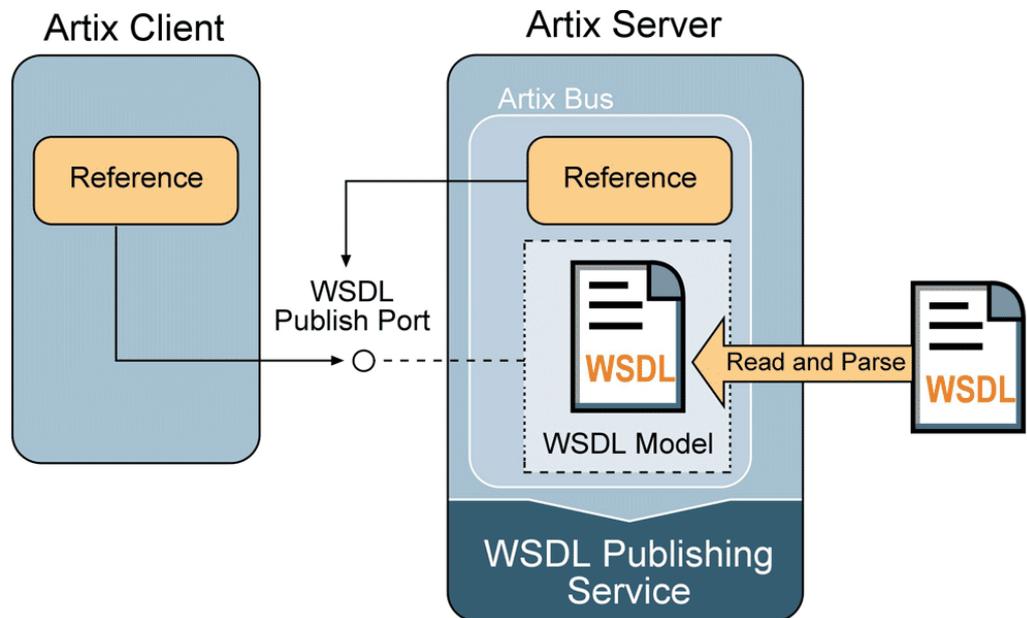
Multiple transports

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

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.

NOTE: If you want to run the WSDL publishing service in a servlet container, refer to Configuring for Use in a Servlet Container.

Configuration file

Example 52 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.

Example 52. Configuring the WSDL Publishing Service

```
❶<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">

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

❸    <bean id="WSDLPublish" class="com.iona.soa.wsdlpublish.rt.WSDLPublish">
❹       <property name="publishPort" value="27220"/>
❺       <property name="publishHostname" value="myhost"/>
❻       <property name="catalogFacility" value="true"/>
❼       <property name="processWSDL" value="standard"/>
❽       <property name="removeSchemas" ref="rschemas"/>
    </bean>
❾    <bean id="rschemas" class="com.iona.cxf.wsdlpublish.Valuelist"
value="http://cxf.apache.org/ http://schemas.iona.com/"/>

</beans>
```

The configuration shown in Example 52 can be explained as follows:

❶ Includes an opening Spring `beans` element that declares the namespaces and schema files for the child elements that are encapsulated by the `beans` element.

❷ Specifies the `com.iona.soa.wsdlpublish.rt.WSDLPublishManager` class, which implements the WSDL publishing service manager. The WSDL publishing service manager enables the WSDL publishing service.

❸ Specifies the `com.iona.soa.wsdlpublish.rt.WSDLPublish` class, which implements the WSDL publishing service.

❹ The publishPort property specifies the TCP/IP port on which the WSDL files are published.

❺ The publishHostname property specifies the hostname on which the WSDL publishing service is available.

❻ The catalogFacility property specifies that the catalog facility is enabled.

❼ The processWSDL property specifies the type of processing that is done on the WSDL file before the WSDL file is published.

❽ 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.

❾ 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 17 describes each of the WSDL publishing service configuration options.

Table 17. 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 XML Schema files (as opposed to on a file system). The default value is `true`.

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 value list 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/.

Configuring for Use in a Servlet Container

You can run the Artix WSDL publishing service in a servlet container, such as Tomcat. This section assumes that you already know how to deploy and run Artix applications in a servlet container. If not, please refer to *Deploying to a Servlet Container.*

Configuration steps

To configure the Artix WSDL publishing service to run in a servlet container, such as Tomcat, complete the following steps:

• Create a spring.xml configuration file

• Configure a listener class in the web.xml file

Create a spring.xml configuration file

Create a `spring.xml` configuration file as shown in Example 53 and include it in the `WEB-INF` directory of your application WAR file.

Example 53. Configuring Artix WSDL Publish Service for Deployment to a Servlet Container

```
<beansxmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws" xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-http-binding.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

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

  <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>
  <bean id="WSDLPublish" class="com.iona.soa.wsdlpublish.rt.WSDLPublish">
      <property name="deployedInContainer" value="true"/>
  </bean>

</beans>
```

Configure a listener class in the web.xml file

Add the configuration shown in Example 54 to your application's
`web.xml` file. Include the `web.xml` file in the `WEB-INF` directory of
your application WAR file.

Example 54. Configuring a Listener Class

```
<web-app>
...
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>WEB-INF/spring.xml</param-value>
   </context-param>

   <listener>
      <listener-class> org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

</web-app>
```

Querying the WSDL Publishing Service

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

  Appending `?wsdl` to the address returns the WSDL file for the
  published endpoint.

- ?xsd

  Appending `?xsd` to the address returns the schema file for the
  published endpoint.

- ?services

  Appending `?services` to the address returns an HTML
  formatted page with a list of all published endpoints and any
  resolved schemas. The `?services` query is not supported
  when the WSDL publishing service is running in a servlet
  container.

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:

  <code style="color:red">http://localhost:27220/SoapContext2/SoapPort2?xsd=
  schema1.xsd</code>

- Using ?services:

  <code style="color:red">http://localhost:27220?services</code>

  Returns an HTML page that lists all documents associated with active services.

Example query syntax when running in a servlet container

The following is an example of the query syntax that you can use to query the WSDL publishing service when it is running in a servlet container. The examples shown refer to Tomcat running on port `8080`:

- Using `?wsdl`:

  ```
  http://host/8080/services/servicename?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://host/8080/services/servicename?xsd=schema1.xsd
  ```

NOTE: `services?` is not supported when WSDL publishing service is running in a servlet container.

Appendix
Artix Binding IDs

Table A.1. Binding IDs for Message Bindings

Binding	ID
CORBA	`http://cxf.apache.org/bindings/corba`
HTTP/REST	`http://apache.org/cxf/binding/http`
SOAP 1.1	`http://schemas.xmlsoap.org/wsdl/soap/http`
SOAP 1.1 w/ MTOM	`http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true`
SOAP 1.2	`http://www.w3.org/2003/05/soap/bindings/HTTP/`
SOAP 1.2 w/ MTOM	`http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true`
XML	`http://cxf.apache.org/bindings/xformat`