PROGRESS SOFTWARE

# Artix® ESB

## Artix® ESB Deployment Guide

Version 5.5
December 2008

# Artix® ESB Deployment Guide

Version 5.5

Publication date 15 Jan 2010


***Legal Notices***

# Table of Contents

# List of Figures

# List of Tables

# List of Examples

# Preface

# What is Covered in This Book

This book explains how to configure and deploy Artix ESB Java Runtime services and applications, including those written in JAX-WS and JavaScript. For details of using Artix® ESB in a C++ or JAX-RPC environment, see Configuring and Deploying Artix Solutions, C++ Runtime[1].

[1] ../cpp/index.htm

# Who Should Read This Book

The main audience of this book is Artix ESB 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 ESB Configuration Overview* on page 17 describes Artix Java configuration files.
- *Setting Up Your Environment* on page 23 describes how to set up your Artix Java environment.
- *Configuring Artix ESB Endpoints* on page 29 describes how to configure Artix Java endpoints.
- *Artix ESB Logging* on page 45 describes how to use logging.
- *Deploying to the Spring Container* on page 77 describes how to deploy an Artix Java endpoint to the Spring container.
- *Deploying to a Servlet Container* on page 97 describes how to deploy an Artix Java endpoint to a servlet container.
- *Deploying WS-Addressing* on page 115 describes how to configure Artix Java endpoints to use WS-Addressing.
- *Enabling Reliable Messaging* on page 123 describes how to enable and configure Web Services Reliable Messaging (WS-RM)..
- *Enabling High Availability* on page 145 describes how to enable and configure both static failover and dynamic failover.
- *Publishing WSDL Contracts* on page 157 describes how to enable the Artix Java WSDL publishing service.
- *Accessing Services Using UDDI* on page 167 which describes how to configure a client to access a WSDL contract from a UDDI registry at runtime.

# The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see Using the Artix ESB Library[2].

[2] http://www.iona.com/support/docs/artix/5.5/library_intro/index.htm

# Artix ESB Configuration Overview

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



Artix ESB 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 ESB 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 ESB and is the recommended approach to use.

# Artix ESB Configuration Files

## Overview

Artix ESB 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 ESB specific elements.

## Spring framework

Spring is a layered Java/J2EE application framework. Artix ESB 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*[1].

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

## Configuration namespace

The core Artix ESB configuration elements are defined in the http://cxf.apache.org/jaxws namespace. You must add the entry shown in Example 1 on page 18 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 namspaces 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 ESB's configuration schemas to the list of schemas in the attribute as shown in Example 2 on page 19.

[1] http://static.springframework.org/spring/docs/2.0.x/reference/beans.html

***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 on page 19 shows a simplified example of a Artix ESB configuration file.

***Example 3. Artix ESB 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 on page 19:

❶ A Artix ESB 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 ESB 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 ESB specific elements

- Plain Spring XML `bean` elements

# Making Your Configuration File Available

**Overview**

You can make the configuration file available to the Artix ESB 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 ESB 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 ESB runtime system environment.*

# Using the Artix ESB Environment Script

**Overview**

To use the Artix ESB runtime environment, the host computer must have several environment variables set. These variables can be configured either during installation, or later using the **fuse_env** script, The can also be configured manually.

**Running the fuse_env script**

The Artix ESB installation process creates a script named **fuse_env**, which captures the information required to set your host's environment variables. Running this script configures your system to use the Artix ESB runtime. The script is located in the *InstallDir*/bin folder.

# Artix ESB Environment Variables

## Overview

This section describes the following environment variables in more detail:

- CXF_HOME
- JAVA_HOME
- ANT_HOME
- SPRING_CONTAINER_HOME
- CATALINA_HOME
- PATH

### 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 **fuse_env** script.

## Environment variables

The environment variables are explained in Table 1 on page 25.

***Table 1. Artix ESB Environment Variables***

| Variable | Description |
|---|---|
| CXF_HOME | Specifies the top level of your Artix ESB installation. For example, on Windows, if you install Artix ESB into the C:\Artix directory, CXF_HOME should be set to C:\Artix. |
| JAVA_HOME | Specifies the directory path to your system's JDK. |
| ANT_HOME | Specifies the directory path to the **ant** utility. The default location is *InstallDir*\tools\ant. |
| SPRING_CONTAINER_HOME | Specifies the directory path to the Spring container. The default location is CXF_HOME\containers\spring_container. |
| CATALINA_HOME | Specifies the location of the Tomcat instance installed with Artix ESB. |

| Variable | Description |
| --- | --- |
| `PATH` | The Artix ESB `bin` directories are prepended on the `PATH` to ensure that the proper libraries, configuration files, and utility programs are used. |

# Customizing your Environment Script

**Overview**

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

The **fuse_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 **fuse_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 **fuse_env** script, it also exits without completing.

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

**Environment variables**

The following applies to the environment variables set by the **fuse_env** script:

- `JAVA_HOME` defaults to the value obtained from the installer. If you do not manually set this variable before running **fuse_env**, it takes its value from the installer.
- The following environment variables are all set with default values relative to `CXF_HOME`:
  - `ANT_HOME`
  - `SPRING_CONTAINER_HOME`
  - `CATALINA_HOME`

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

# Configuring Artix ESB Endpoints

*Artix ESB 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 ESB 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

**Overview**

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 on page 32 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*. |
| `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 Appendix A on page 171. |

| 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 ESB APIs, such as `Endpoint.publish()` or `Service.getPort()`.<br>The default is `false`.<br>Setting this to `true` does the following:<br>• Changes the internal name of the bean by appending `.jaxws-endpoint` to its id<br>• Makes the bean abstract |

In addition to the attributes listed in Table 2 on page 32, you might need to use multiple `xmlns:`*shortName* attributes to declare the namespaces used by the `endpointName` and `serviceName` attributes.

**Example**

Example 4 on page 33 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 on page 34 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

**Overview**

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 a Artix ESB specific object. If you are using a pure Spring approach to building your services, you will not be forced to use Artix ESB 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 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 on page 36 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. |
| `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*. |
| `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 Appendix A on page 171. |

| 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`, 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 ESB APIs, such as `Endpoint.publish()` or `Service.getPort()`.<br>The default is `false`.<br>Setting this to `true` does the following:<br>• Changes the internal name of the bean by appending `.jaxws-endpoint` to its id<br>• Makes the bean abstract |

In addition to the attributes listed in Table 3 on page 36, you might need to use multiple `xmlns:`*shortName* attributes to declare the namespaces used by the `endpointName` and `serviceName` attributes.

**Example**

Example 6 on page 37 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

**Overview**

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 on page 38 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*. |
| `jaxws:inInterceptors` | Specifies a list of interceptors that process inbound requests. For more information see Developing Interceptors for the Java Runtime. |
| `jaxws:inFaultInterceptors` | Specifies a list of interceptors that process inbound fault messages. For more information see Developing Interceptors for the Java Runtime. |
| `jaxws:outInterceptors` | Specifies a list of interceptors that process outbound replies. For more information see Developing Interceptors for the Java Runtime. |
| `jaxws:outFaultInterceptors` | Specifies a list of interceptors that process outbound fault messages. For more information see Developing Interceptors for the Java Runtime. |
| `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.[a] |
| `jaxws:dataBinding`[b] | Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition. |

| Element | Description |
|---|---|
| `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 ESB. You can provide either a list of bean references or a list of embedded beans. |
| `jaxws:invoker` | Specifies an implementation of the `org.apache.cxf.service.Invoker` interface used by the service. [c] |
| `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. |

[a]The SOAP binding is configured using the `soap:soapBinding` bean.

[b]The `jaxws:endpoint` element does not support the `jaxws:dataBinding` element.

[c]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.

# Configuring Consumer Endpoints

**Overview**

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 on page 40 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 Appendix A on page 171. |
| `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). |

| Attribute | Description |
|---|---|
| `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 ESB APIs like `Service.getPort()`.<br>The default is `false`.<br>Setting this to `true` does the following:<br>• Changes the internal name of the bean by appending `.jaxws-client` to its id<br>• Makes the bean abstract |

In addition to the attributes listed in Table 5 on page 40, 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 on page 42 describes the child element's you can use to configure a 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.[a] |
| `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 ESB. 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*. |
| `jaxws:inInterceptors` | Specifies a list of interceptors that process inbound responses. For more information see Developing Interceptors for the Java Runtime. |
| `jaxws:inFaultInterceptors` | Specifies a list of interceptors that process inbound fault messages. For more information see Developing Interceptors for the Java Runtime. |
| `jaxws:outInterceptors` | Specifies a list of interceptors that process outbound requests. For more information see Developing Interceptors for the Java Runtime. |
| `jaxws:outFaultInterceptors` | Specifies a list of interceptors that process outbound fault messages. For more information see Developing Interceptors for the Java Runtime. |
| `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. |

[a]The SOAP binding is configured using the `soap:soapBinding` bean.

**Example**

Example 7 on page 43 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 ESB Logging

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

# Overview of Artix ESB Logging

**Overview**

Artix ESB 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 ESB 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 ESB includes a logging feature that can be plugged into your client or your service to enable logging. Example 8 on page 46 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 on page 57.

**Where to begin?**

To run a simple example of logging follow the instructions outlined in a Simple Example of Using Logging on page 48.

For more information on how logging works in Artix ESB, 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 documents gives a good overview of `java.util.logging`:

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

- http://java.sun.com/j2se/1.5.0/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=%CXF_HOME%\etc\logging.properties demo.hw.server.Server`** |
| UNIX | **`java -Djava.util.logging.config.file=$CXF_HOME/etc/logging.properties demo.hw.server.Server &`** |

   The default `logging.properties` file is located in the *`InstallDir`*`/etc` directory. It configures the Artix ESB 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=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server` |
| UNIX | `java -Djava.util.logging.config.file=$CXF_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.properties` File



The default logging configuration file, `logging.properties`, is located in the *`InstallDir`*`/etc` directory. It configures the Artix ESB 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: http://java.sun.com/j2se/1.5/docs/api/java/util/logging/package-summary.html.

# Configuring Logging Output

The Java logging utility, `java.util.logging`, uses handler classes to output log messages. Table 7 on page 51 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 ESB environment. For details on setting the Artix ESB environment, see Using the Artix ESB Environment Script on page 24.

### Configuring the console handler

Example 9 on page 51 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 on page 51.

***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 51 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 on page 53). 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 on page 52 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 on page 52.

***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 on page 52 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 on page 52.

***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 on page 53.

***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 on page 53 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

**Overview**

You can run the logging utility on an application by defining a `java.util.logging.config.file` property when you start the application. You can 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 on page 54 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 on page 53 to set fine-grained logging for specified Artix ESB logging subsystems.

## Artix ESB logging subsystems

Table 8 on page 55 shows a list of available Artix ESB logging subsystems.

***Table 8. Artix ESB Logging Subsystems***

| Subsystem | Description |
|---|---|
| `com.iona.cxf.container` | Artix ESB container |
| `org.apache.cxf.aegis` | Aegis binding |
| `org.apache.cxf.binding.coloc` | colocated binding |
| `org.apache.cxf.binding.http` | HTTP binding |
| `org.apache.cxf.binding.jbi` | JBI 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 ESB 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.jbi` | JBI container integration classes |
| `org.apache.cxf.jca` | JCA container integration classes |
| `org.apache.cxf.js` | JavaScript front-end |
| `org.apache.cxf.transport.http` | HTTP transport |
| `org.apache.cxf.transport.https` | secure version of HTTP transport, using HTTPS |

| Subsystem | Description |
|---|---|
| `org.apache.cxf.transport.jbi` | JBI transport |
| `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/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 on page 56.

***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 on page 56 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 on page 57.

***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 on page 57 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 on page 57.

***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 on page 57 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 on page 58.

***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 on page 58 to the `cxf.xml` configuration file located 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=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server`** |
| UNIX | **`java -Djava.util.logging.config.file=$CXF_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=%CXF_HOME%\etc\mylogging.properties demo.hw.client.Client .\wsdl\hello_world.wsdl`** |
| UNIX | **`java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl`** |

The SOAP messages are logged to the console.

# Deploying to an OSGi Container

*Applications developed using the Artix ESB Java Runtime can be installed into an OSGi container. Once installed the applications can use many of the advanced Artix features.*

# Introduction to OSGi

**Overview**

The adoption of OSGi is a key strategy for Progress Software's SOA portfolio. OSGi is a mature, lightweight, component system that solves many challenges associated with medium and large scale development projects. Through the use of bundles complexity is reduced by separating concerns and ensuring dependencies are minimally coupled via well defined interface communication. This also promotes the reuse of components in much the same way that SOA promotes the reuse of services. And, since each bundle effectively is given an isolated environment, and since dependencies are explicitly defined, versioning and dynamic updates are possible. These are just a few of the many benefits of OSGi. Users wishing to learn more should check out http://www.osgi.org/Main/HomePage and see for yourself why Progress Software recommends the use of OSGi in your projects.

Before you can install an application into an OSGi container, you need to package it into one or more *OSGi bundles*. An OSGi bundle is a JAR that contains extra information that is used by the OSGi container. This extra information specifies the packages this bundle exposes to the other bundles in the container and any packages on which this bundle depends.

**Supported containers**

Artix Java applications should work in any OSGi container. However, they are only supported in the following containers:

- FUSE ESB 4.0
- Apache ServiceMix 4.0.0.3
- Equinox 3.4.0

Artix includes the Equinox 3.4.0 container. The build files for a selection of the sample applications include Ant targets for setting up and launching the included Equinox runtime.

If you would rather use FUSE ESB you can download it from http://fusesource.com.

If you would rather use Apache ServiceMix you can download it from http://servicemix.apache.org/SMX4.

**Supported features**

All of the the core functionality of the Artix ESB Java Runtime is available in an OSGi container. A number of the enterprise features are also available including:

- the CORBA transport
- the advanced WSDL publishing features
- working with the Artix locator
- static and dynamic fail-over
- security

### Application packaging

Before you can install an application into an OSGi container it needs to be packaged into one or more OSGi bundles. An OSGi bundle is essentially a standard JAR. Where an OSGi bundle and a plain JAR differ is in the contents of their manifest files. An OSGi bundle's manifest contains a number of properties that specify the following:

- the Java packages which this bundle exposes to other bundles
- the Java packages, and other resources, on which this bundle depends
- the version number of the bundle

### Bundle life-cycles

Applications in an OSGi environment are subject to the life-cycle of its bundles. Bundles have six life-cycle states:

Installed
: All bundles start in the installed state. Bundles in the installed state are waiting for all of their dependencies to be resolved. Once all of the bundle's dependencies are resolved it moves to the next life-cycle state.

Resolved
: Bundles are moved into the resolved state when the following conditions are met:

  - The runtime environment meets or exceeds the one specified by the bundle.
  - All of the packages imported by the bundle are exposed by bundles that are either in the resolved state or that can be moved into the resolved state at the same time as the current bundle.
  - All of the required bundles are either in the resolved state or can be resolved at the same time as the current bundle.

## Important

All of an application's bundles must be in the resolved state before it can be started.

If, at any time, one of the above conditions ceases to be satisfied the bundle is moved back into the installed state. This can happen if a bundle containing an imported package is removed from the container.

Starting
: The starting state is a transitory state between the resolved state and the active state. When a bundle is started, the container needs to create the resources for the bundle. The container will also call the `start()` method of the bundle's bundle activator if one is provided.

Active
: Bundles in the active state are available to do work. What a bundle does in the active state depends on the contents of the bundle. For a bundle containing a JAX-WS service provider this means the service is available to accept requests.

Stopping
: The stopping state is a transitory state between the active state and the resolved state. When a bundle is stopped, the container needs to clean up the resources for the bundle. The container will also call the `stop()` method of the bundle's bundle activator if one is provided.

Uninstalled
: When a bundle is uninstalled it is moved from the resolved state to the uninstalled state. Bundles in this state cannot be transitioned back into the resolved state or any other state. It must be explicitly re-installed.

For application developers the important life-cycle states are the starting state and the stopping state. The endpoints exposed by your application need to be published during in the start state. The published endpoints need to be stopped during the stopping state.

# Packaging and Installing an Application

**Overview**

Artix comes with several sample applications that show how to package and install applications into an OSGi container. While there are a number of tools available and the different OSGi containers use varying deployment mechanisms the basic steps are the same:

1. Determine how the application will publish its endpoints.

   In order for your application to be started by the OSGi framework it needs to have logic that instantiates the resources used by the application. For a JAX-WS server application this means that the service provider endpoints need to be created and published. For a JAX-WS client application this means that the object implementing the client's business logic needs to be instantiated. The application may also need some logic to clean up its resources when shutting down.

   There are two ways to get the resources for your application started:

   - rely on Sring-DM to publish the endpoints
   - publish the endpoints in a `BundleActivator` object

2. Decide how you want to bundle the application.

   Depending on the complexity of you application, it may make sense to break your application into several bundles. You will also want to be very deliberate in determining if your application's bundles include any third party libraries.

   **Tip**

   The best practice is to find OSGi bundles containing all of the third party libraries needed by your application and use the OSGi dependency mechanism to resolve them.

3. Create bundles for each of your application's modules.
4. Install the bundles to the container.

Once your bundles are installed, you can start and stop them using the container's commands. You can also upgrade them in place when needed.

### Publishing endpoints using Spring-DM

Spring Dynamic Modules for OSGi Service Platforms(Spring-DM) scans bundles in an OSGi container and automatically creates a Spring application context for any bundles that contains a Spring XML file in its `META-INF/spring` folder. It will inject all of the beans defined in the configuration file into the application context. For JAX-WS server, you would include a `jaxws:endpoint` element for each service provider your application exposes. Spring-DM will inject them into the application context and publish the endpoints when the bundle is started.

Before using Spring-DM you need to make sure the required bundles are installed into your OSGi container. You also need to make sure they are started. The demos included with Artix ESB configure the included Equinox container to install and start the Spring-DM bundles. FUSE ESB and Apache ServiceMix include support for Spring-DM as part of their default configuration. If your container does not include Spring-DM support you can down load it from http://www.springsource.org/osgi.

#### Important

When using Spring-DM to publish endpoints you must include the following DynamicImport-Package statement in the bundle's manifest:

```
DynamicImport-Package: org.apache.cxf.*
```

This allows the Spring framework to construct the proper Artix ESB objects.

### Publishing endpoints in a bundle activator

The `org.osgi.framework .BundleActivator` is an OSGi API used by the OSGi framework when it starts and stops bundles. As shown in Example 22 on page 66, it has two methods that need to be implemented.

***Example 22. Bundle Activator Interface***

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

The `start()` method is called when the container starts the bundle. This is where you would instantiate and publish any service provider endpoints exposed by a server application. For a client application, this is the method that would instantiate the client's application logic.

The `stop()` method is called when the container stops the bundle. This is where you would stop any endpoints exposed by a server.

When using a bundle activator you need to add the Bundle-Activator property to the bundle's manifest. This property tells the container which class in the bundle to use when activating the bundle.

For more information on publishing service provider endpoints see *Publishing a Service* in *Developing Artix® Applications with JAX-WS*.

### Creating bundles

You can hand edit the manifest files for each of your application's bundles and manually assemble them using the standard Java mechanism for creating JARs. Hand editing a manifest is a tedious and error prone process and is not recommended. You should use one of the many tools available for creating OSGi bundles.

Artix includes the **bnd**[1] tool from Peter Kriens. It automates the construction of OSGi bundle manifests by introspecting the contents of the classes being packaged in the bundle. Using the knowledge of the classes contained in the bundle, the tool can calculate the proper values to populate the Import-Packages and the Export-Package properties in the bundle manifest.

The **bnd** tool can be used as an Ant task and is the foundation for the Maven bundle plug-in[2] from Apache Felix.

### Required bundles

The Artix ESB Java Runtime core components are included in an OSGi bundle called `org.apache.cxf.cxf-bundle`. The advanced features are packaged in the bundles listed in Table 9 on page 67.

***Table 9. Advanced Feature Bundles***

| Feature | Bundles |
|---|---|
| Locator Service Discovery Support | `it-soa-ha-common-api`, `it-soa-ha-discovery-client-locator-rt`, `it-soa-common-rt` |

[1] http://www.aqute.biz/Code/Bnd
[2] http://cwiki.apache.org/FELIX/apache-felix-maven-bundle-plugin-bnd.html

| Feature | Bundles |
|---|---|
| High Availability | `it-soa-ha-common-api`, `it-soa-ha-discovery-client-locator-rt`, `it-soa-ha-failover-locator-rt`, `it-soa-common-rt` |
| WSDL Publishing | `it-soa-wsdlpublish-api`, `it-soa-wsdlpublish-rt`, `it-soa-common-rt` |
| Security | `it-soa-security-bundle`, `it-soa-common-rt` |
| CORBA | `it-soa-bindings-corba-rt`, `it-soa-common-rt` |
| FTP Transport | `it-soa-transports-ftp-api`, `it-soa-transports-ftp-rt`, `it-soa-common-rt` |

Before you can launch your application, all of the Artix ESB bundles required by your application must be installed into the container.

**Tip**

The Equinox configuration file generated by the Ant target included in the OSGi demos automatically loads the `org.apache.cxf.cxf-bundle` bundle.

**Tip**

FUSE ESB and Apache ServiceMix install the `org.apache.cxf.cxf-bundle` bundle by default.

### Required packages

Artix ESB Java Runtime applications depend on a number of runtime packages. Because of the complex nature of the dependencies in Artix, you cannot rely on the **bnd** tool to automatically determine the needed imports. You will need to explicitly declare them.

You need to import the following packages into your bundle:

- `javax.jws`
- `javax.wsdl`
- `META-INF.cxf`
- `META-INF.cxf.osgi`

- `org.apache.cxf.bus`
- `org.apache.cxf.bus.spring`
- `org.apache.cxf.bus.resource`
- `org.apache.cxf.configuration.spring`
- `org.apache.cxf.resource`
- `org.springframework.beans.factory.config`

# Installing a Sample Application

### The WSDL-first example

This section is going to use the WSDL-first sample included with Artix to walk you through installing an application into an OSGi container. This sample, which is located in *InstallDir*/java/samples/basic/wsdl-first, is a simple Hello World application that is developed using the JAX-WS APIs.

Its build file contains two Ant targets for installing the sample into the included Equinox OSGi container:

- **osgi**—Packages the application into two bundles.
- **start.equinox**—Generates a configuration file listing the bundles required by the demo and starts the Equinox container in console mode.

Once you have started the container and built the bundles, you can install them into the running container and see the client make requests against the service. Once you are finished testing the example you can stop the application and uninstall it.

This section is going to focus on the **osgi** Ant target, the process of installing the application into the running container, and the process of uninstalling the application.

### Building the bundles

Example 23 on page 70 shows the Ant targets used to package the WSDL-first demo into two bundles.

***Example 23. Build Targets for Packaging the Demo as OSGi Bundles***

```
<target name="osgi" depends="build">  ❶
  <antcall target="osgi.endpoint"/>
  <antcall target="osgi.jaxb.model"/>
</target>

<property name="bundle.model.version" value="1.0"/> ❷
<property name="bundle.endpoint.version" value="1.0"/>

<target name="osgi.endpoint" depends="build"> ❸
  <echo level="info" message="Generating CXF Endpoint OSGi bundle..."/>
  <cxfbnd private-package="demo.hw.server.impl;version=${bundle.endpoint.version},
!org.apache.hello_world_soap_http.*" ❹
              import="${default.cxf.osgi.import}"
              include-resource="META-
INF/spring/=spring/osgi/,hello_world.wsdl=wsdl/hello_world.wsdl"
              bundle="${cxf.war.file.name}_endpoint"
```

```
              version="${bundle.endpoint.version}"/>
</target>

<target name="osgi.jaxb.model" depends="build"> ❺
  <echo level="info" message="Generating CXF JAXB Model OSGi bundle..."/>
  <cxfbnd import="*"
              export="org.apache.hello_world_soap_http.*;version=${bundle.model.version}"

              bundle="${cxf.war.file.name}_model"
              version="${bundle.model.version}"/>
</target>
```

The Ant targets in Example 23 on page 70 do the following:

❶ Calls the targets that are responsible for building the actual bundles.

❷ Sets the name and version of the bundles as properties.

❸ Packages the service implementation into an OSGi bundle.

❹ Calls the **cxfbnd** Ant marco to invoke the **bnd** tool.

The **cxfbnd** Ant marco is in the *InstallDir*/java/sample/common_build.xml file.

❺ Packages the the JAXB classes used by the service implementation into an OSGi bundle.

This target also uses the **cxfbnd** Ant marco.

The **cxfbnd** Ant macro builds a BND driver file similar to the one shown in Example 24 on page 71. It then calls the **bnd** tool to package the specified classes into an OSGi bundle.

***Example 24. OSGi BND Control File***

```
Private-Package: demo.hw.server.impl;version=1.0, !org.apache.hello_world_soap_http.*
Import-Package: javax.jws, javax.wsdl, META-INF.cxf, org.apache.cxf.bus,
org.apache.cxf.bus.spring, org.apache.cxf.bus.resource, org.apache.cxf.configuration.spring,
 org.apache.cxf.resource, org.springframework.beans.factory.config, *
Include-Resource: META-INF/spring/=spring/osgi/,hello_world.wsdl=wsdl/hello_world.wsdl
Bundle-Version: 1.0
Require-Bundle: org.apache.cxf.cxf-bundle
DynamicImport-Package: org.apache.cxf.*
```

To package the demo as OSGI bundles you would enter the following:

```
>ant osgi
```

When Ant finishes building and packaging the demo you will find the following bundles in the `build` folder:

- `helloworld_endpoint-1.0.jar`—the bundle holding the service implementation
- `helloworld_model-1.0.jar`—the bundle holding the JAXB objects used by the service implementation

## Note

The demo splits the application into two bundles merely to illustrate that application can be partitioned into multiple bundles. There is no requirement that an application be split into multiple bundles. In fact, unless another application required the JAXB model used by the HelloWorld application, it would be neater to package this application as a single bundle.

### Installing the application

Once you have packaged your application into bundles, you must install them to a running OSGi container. How you install bundles into your OSGi container will depend on your OSGi container.

If you are using the included Equinox OSGi container you use the **install** command as shown in Example 25 on page 72.

***Example 25. Installing HelloWorld to the Equinox OSGi Container***

```
osgi>install InstallDir/java/samples/basic/wsdl_first/build/helloworld_endpoint-1.0.jar
Bundle id is 58
[Framework Event Dispatcher] INFO helloworld_endpoint-1.0 - BundleEvent INSTALLED
osgi>install InstallDir/java/samples/basic/wsdl_first/build/helloworld_model-1.0.jar
Bundle id is 59
[Framework Event Dispatcher] INFO helloworld_model-1.0 - BundleEvent INSTALLED
```

## Important

You need the bundle IDs to work with the bundles once they are installed in the container. The IDs are the only means of addressing specific bundles.

When the bundles are installed they are placed into the installed life-cycle state. The Equinox container does not automatically move them into the resolved state.

In this state the bundles cannot do anything. The container knows that the bundles are loaded and that their contents are available to bundles that need them.

### Starting the application

Before an application can be used the bundle containing the service implementation must be moved into the active life-cycle state and all of the bundles containing dependencies must be in the resolved, or active, life-cycle state.

In the demo, the service implementation is packaged in the `helloworld_endpoint-1.0` bundle. The bundle contains a Spring configuration file that defines the endpoint to be created by the application. The Equinox configuration generated for the demo installs and starts the Spring-DM bundles. The Spring-DM extender bundle creates a Spring `ApplicationContext` for the `jaxws:endpoint` element included in the endpoint bundle's `META-INF/spring` directory. For more information about writing Spring configuration for a service provider see Configuring Service Providers on page 30.

## Note

FUSE ESB and Apache ServiceMix 4 also come preloaded with the Spring-DM extender bundle.

To start the demo application you use the container's **start** command as shown in Example 26 on page 73.

***Example 26. Starting HelloWorld in the Equinox OSGi Container***

```
osgi>start 58
```

## Tip

The start command uses the bundle ID of the bundle you wish to start as its argument.

When you issue the **start** command for the endpoint bundle, the container begins trying to resolve all of the bundle's dependencies. Because the endpoint bundle depends on packages from the model bundle, the container automatically resolves the `helloworld_model-1.0` bundle before resolving the endpoint bundle. Once the endpoint's bundle's dependencies are resolved,

the container activates it. This leaves the model bundle in the resolved life-cycle state and the endpoint bundle in the active life-cycle state.

### Running the client

Once the endpoint bundle is fully started, the HelloWorld service provider is ready to accept requests from consumers. To test the application you can run the client against the service provider by executing the **ant client** command. Example 27 on page 74 shows the expected results.

***Example 27. Output from HelloWorld***

```
Buildfile: build.xml

validate.thirdparty.classpath:

maybe.generate.code:

compile:

create.spring.war:

demo.build:

build:

client:

validate.thirdparty.classpath:
     [java] file:/C:/iona/artix_5.5/java/samples/basic/wsdl_first/wsdl/hello_world.wsdl
     [java] Invoking sayHi...
     [java] Server responded with: Bonjour
     [java]
     [java] Invoking greetMe...
     [java] Server responded with: Hello Finn
     [java]
     [java] Invoking greetMe with invalid length string, expecting exception...
     [java] Caught expected WebServiceException:
     [java]     Marshalling Error: cvc-maxLength-valid: Value 'Invoking greetMe with invalid length
string, expecting exception...
' with length = '67' is not facet-valid with respect to maxLength '30' for type 'MyStringType'.
     [java]
     [java] Invoking greetMeOneWay...
     [java] No response from server as method is OneWay
     [java]
     [java] Invoking pingMe, expecting exception...
     [java] Expected exception: PingMeFault has occurred: PingMeFault raised by server
     [java] FaultDetail major:2
     [java] FaultDetail minor:1
```

```
BUILD SUCCESSFUL
Total time: 6 seconds
```

### Stopping the application

When you are ready to take the service provider off line you can simply move it from the active state to the resolved state. This will shut down the service provider and free the resources it is using. However, the contents of the application's bundle remain available to other bundles.

To stop a bundle in Equinox you use the **stop** command.

#### Tip

Stopped bundles can be easily reactivated using the **start** command.

### Uninstalling the application

When you want to completely remove any resources exposed by your application's bundles you need to uninstall the bundles. Once a bundle is uninstalled none of its exported resources are available to bundles in the container. It also cannot be re-started. It must be reinstalled before it can be used again.

To uninstall bundles from the Equinox container you use the **uninstall** command.

# Deploying to the Spring Container

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

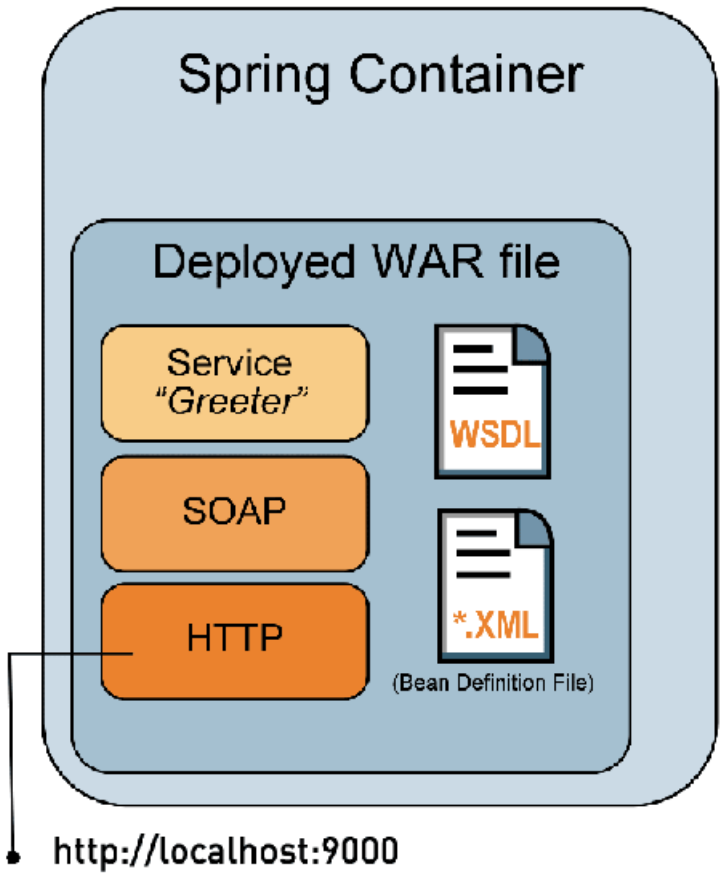
# Introduction

## Overview

Artix ESB 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 ESB service endpoints in the Spring container.

Figure 1 on page 79 shows how to access a deployed Artix ESB endpoint in the Spring container.

*Figure 1. Artix ESB 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 a Artix ESB 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

Most of the samples contained in the *InstallDir*/samples directory can be deployed to the Spring container. After reading this chapter, you can try deploying some of the sample applications to the Spring container. For instructions, see the README.txt files in each sample directory.

# Running the Spring Container

**Overview**

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 10. Spring Container Command Options***

| | |
|---|---|
| -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.<br><br>By default, the Spring container uses the spring_container.xml file, which is located in the *InstallDir*/containers/spring_container/etc directory.<br><br>You should only use the -config flag if you are specifying a different configuration file. For example, see Running Multiple Containers on Same Host on page 93 . |
| -wsdl *container-wsdl-url* | Specifies the URL or file location of the Spring container WSDL file. This flag is optional.<br><br>By default, the Spring container uses the container.wsdl file located in the *InstallDir*/containers/spring_container/etc/wsdldirectory.<br><br>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 on page 93 |
| -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 that one container on a single host, see Running Multiple Containers on Same Host on page 93.

### 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 on page 93.

# Deploying a Artix ESB Endpoint

## Deployment steps

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

1. Write a Artix ESB configuration file for your application. See Configuring your application on page 83.
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 on page 84.
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 on page 85.
   - Use the JMX console. See Managing the Container using the JMX Console on page 86.
   - Use the Web service interface. See Managing the Container using the Web Service Interface on page 89.

## 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 28 on page 83 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. Most of the samples in the *`InstallDir`*`/samples` directory contain files named`spring.xml`, which configure the samples for deployment to the Spring container.

***Example 28. 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 28 on page 83 can be explained as follows:

❶ The Spring `beans` element is required at the beginning of every Artix ESB 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.

**Important**

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

For more information on configuring a Artix ESB `jaxws:endpoint` element, see Using the jaxws:endpoint Element on page 31.

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 **artix**

**wsdl2java** utility. For details, see artix wsdl2java in *Artix® ESB 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 a Artix ESB 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 ESB 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 applications, including the WAR file for deployment to the Spring container, see the example `build.xml` file located in the *InstallDir*`/samples/spring_container/wsdl_first` 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 29 on page 90 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 29 on page 90 for more detail.

# Managing the Container using the JMX Console

**Overview**

You can use the JMX console to deploy and manage applications in the Spring container. 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:

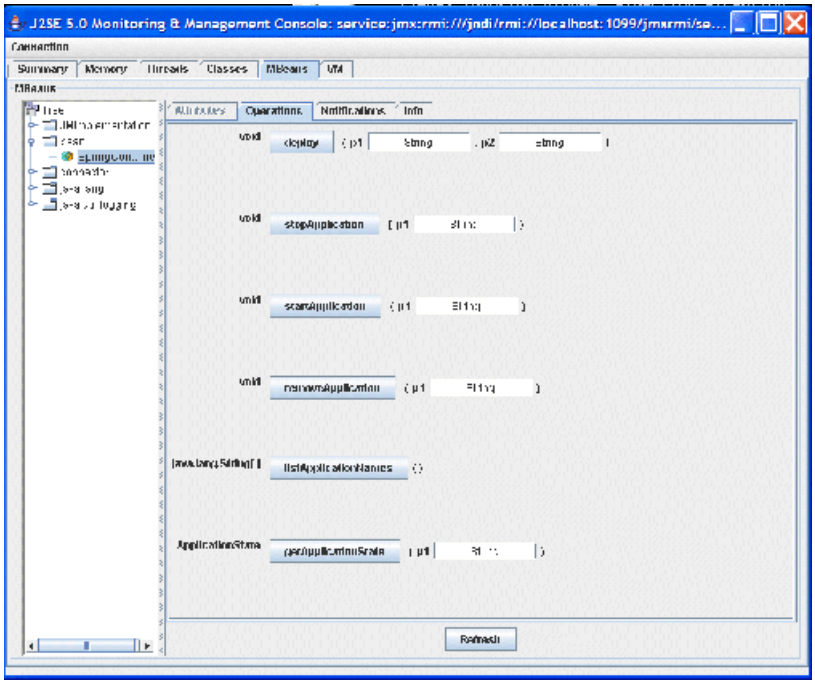
1. Start the JMX console by running the following command from the *InstallDir*`/bin` directory:

| 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 on page 87).

   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 11 on page 87.

*Table 11. 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.<br>`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. |

| Operation | Description | Parameters |
|---|---|---|
| `startApplication` | Starts an application that has previously been deployed and subsequently stopped. | `name` — Specifies the name of the application that you want to start. The application name is the same as the WAR file name. |
| `removeApplication` | Stops and removes an application. It completely removes an application from the container repository. | `name` — Specifies the name of the application that you want to stop and remove. The application name is the same as the WAR file name. |
| `listApplicationNames` | Lists all of the applications that have been deployed. The applications can be in one of three states: start, stop, or failed. An application's name is the same as its WAR file name. | None |
| `getApplicationState` | Reports 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

**Overview**

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

**Client tool**

Artix ESB 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 the Developing Artix® Applications with JAX-WS 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/Ad
minPort" ...>
```

You do not need to change the `container.wsdl` file.

For more information on the `spring_container.xml` file, see Spring Container Definition File on page 90.

**Adding a port**

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

# Spring Container Definition File

**Overview**

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

*InstallDir*`/containers/spring_container/etc`

**spring_container.xml**

The contents of the Spring container configuration file are shown in Example 29 on page 90.

***Example 29. spring_container.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="C:\iona\fuse-services-frame
work/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.ContainerAdminSer
viceImpl">
    <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.JMXContain
er</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 29 on page 90 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 2.0.x reference document available at http://static.springframework.org/spring/docs/2.0.x/reference/jmx.html.

# Running Multiple Containers on Same Host

**Overview**

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`:

   1. Container repository location—change the container's `containerRepository` property to point to a new repository.

      For example, you change:

```
<container:container id="container"
      containerRepository="c:\iona\fuse-services-framework/containers/spring_container/re
pository"
       scanInterval="5000"/>
```

      To:

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

2. 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">
```

3. 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>
```

4. Change the RMI registry port from `1099` to a new port as shown in the following snipit:

```
<bean id="registry" class="org.springframework.remot
ing.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_con
tainer\wsdl\my_container.wsdl stop
```

# Deploying to a Servlet Container

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

# Introduction

## Overview

Servlet containers are a common platform for running Web services. The Artix ESB 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:

- a Artix ESB provided servlet adapter class
- a custom servlet
- the Spring servlet context listener
- the Artix ESB 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 ESB 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 ESB JCA adapter.

For more information on using the JCA adapter read Artix for J2EE (JAX-WS) .

# Configuring the Servlet Container

## Overview

Before you can deploy a Artix ESB endpoint to your servlet container you must make the Artix ESB 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 ESB 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 ESB endpoints require all of the JAR files in the *`InstallDir`*`/lib` directory except the following:

- `cxf-*-jbi-*.jar`
- `ap-*.jar`
- `artix.jar`
- `bnd*.jar`
- `compendium*.jar`
- `derby*`
- `geronimo-ejb*`
- `geronimo-j2ee*`
- `geronimo-servlet*`
- `it-soa-ads*.jar`

- it-soa-broker.jar
- it-soa-container*.jar
- it-soa-jaxwsgenerator*.jar
- it-soa-management*
- it-soa-router.jar
- it-soa-tools
- it-soa-transport-mq*
- osgi-3.4.0.jar
- pax*
- spring-osgi*.jar
- servlet-api*.jar
- geronimo-servlet_*.jar
- jetty-*.jar

### Automating servlet container configuration

The Artix ESB 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 ESB 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.

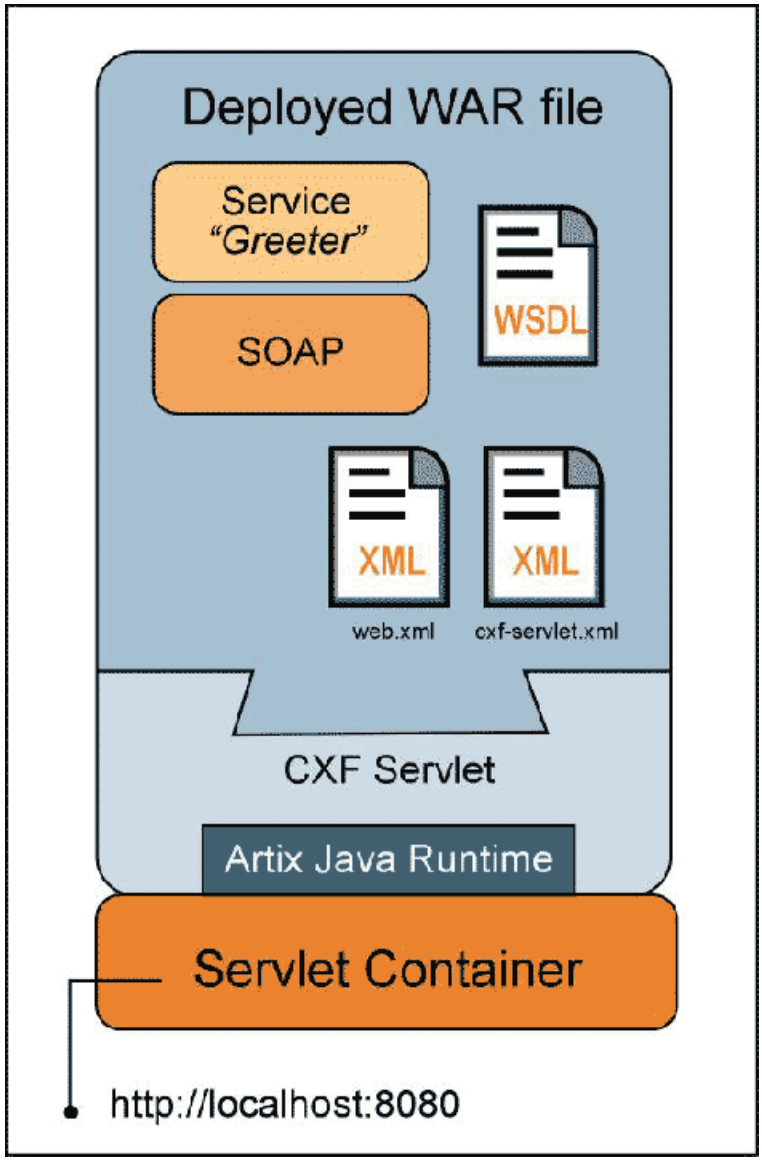
# Using the CXF Servlet

## Overview

Artix ESB 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 on page 102 shows the main components of a Artix ESB endpoint deployed using the CXF servlet.

***Figure 3. Artix ESB 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 ESB configuration file

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

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

---

**Deployment steps**

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

1. Build a WAR that contains your application and all the required support files.
2. 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` — a Artix ESB 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.

## Tip

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 a Artix ESB 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 30 on page 104 shows a simple cxf-servlet.xml file.

***Example 30. CXF Servlet Configuration File***

```
<?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 30 on page 104 is explained as follows:

❶ The Spring `beans` element is required at the beginning of every Artix ESB 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.

**Important**

This class needs to be included in the WAR's `WEB-INF/classes` folder.

- `wsdlLocation` — Specifies the WSDL file that contains the service definition.

**Important**

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 on page 31.

### Web application configuration

You must include a `web.xml` deployment descriptor file that instructs the servlet container to load the CXF servlet. Example 31 on page 106 shows a `web.xml` file. It is not necessary to change this file. A reference copy is located in the *`InstallDir`*`/etc` directory.

***Example 31. A web.xml Deployment Descriptor File***

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

**Deploying 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

## Overview

In some cases, you might want to write a custom servlet that deploys Artix ESB enabled endpoints. A common reason is to deploy Artix ESB 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 a Artix ESB enabled endpoint.
2. Package your servlet in a WAR that contains the Artix ESB 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 ESB 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 ESB endpoint. Example 32 on page 107 shows an example of instantiating a consumer endpoint in a servlet.

***Example 32. 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.

Example 33 on page 108 shows an example of loading the configuration from the `WEB-INF/client.xml` file.

***Example 33. 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 **artix 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

**Overview**

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 ESB components need to be loaded. If any component is missing, the application will not 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 ESB 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 a Artix ESB 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 34 on page 111 shows a `web.xml` file that configures the servlet container to load the Spring listener and a Spring configuration file.

***Example 34. 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 34 on page 111 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 a application using the Spring context listener is similar to a standard Artix ESB configuration file. It uses all of the same endpoint configuration elements described in *Configuring Artix ESB Endpoints* on page 29. It can also contain standard Spring beans.

The difference between a typical Artix ESB 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 ESB 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 35 on page 112 shows the configuration for a simple consumer endpoint being deployed using the Spring context listener.

***Example 35. 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 35 on page 112 import the required Artix ESB component configuration. The required Artix ESB 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 35 on page 112.

**Tip**

Importing the `cxf-all.xml` configuration file will automatically import all of the Artix ESB 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 ESB supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Artix ESB runtime environment.*

# Introduction to WS-Addressing

**Overview**

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 ESB 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

**Overview**

In Artix ESB, WS-Addressing functionality is implemented as interceptors. The Artix ESB 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.

**WS-Addressing Interceptors**

The WS-Addressing implementation consists of two interceptors, as described in Table 12 on page 117.

*Table 12. 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

## Overview

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 ESB 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 36 on page 118 and Example 37 on page 118 respectively.

***Example 36. 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 ht
tp://www.springframework.org/schema/beans/spring-beans.xsd">

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

***Example 37. server.xml—Adding WS-Addressing Feature to Server 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.spring
```

```
framework.org/schema/beans/spring-beans.xsd">

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

# Configuring WS-Addressing Attributes

**Overview**

The Artix ESB WS-Addressing feature element is defined in the namespace http://cxf.apache.org/ws/addressing. It supports the two attributes described in Table 13 on page 120.

*Table 13. 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 38 on page 120 an addressing policy assertion to enable non-anonymous responses is embedded in the `policies` element.

*Example 38. 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 ESB supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in Artix ESB.*

# Introduction to WS-RM

## Overview

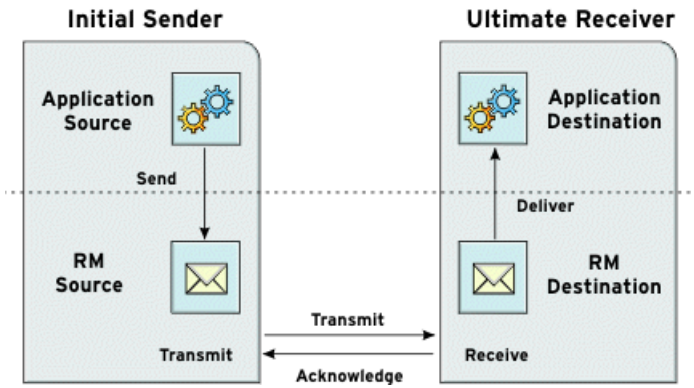
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

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 on page 124.

***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 on page 132.

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 can not be assured.

**Supported specifications**

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

**Further information**

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

# WS-RM Interceptors

## Overview

In Artix ESB, WS-RM functionality is implemented as interceptors. The Artix ESB 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 ESB WS-RM Interceptors

The Artix ESB WS-RM implementation consists of four interceptors, which are described in Table 14 on page 126.

*Table 14. Artix ESB WS-ReliableMessaging Interceptors*

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

## Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, 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 on page 128.

### Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Artix ESB 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 on page 132.

# Enabling WS-RM

**Overview**

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 ESB 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 ESB 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 39 on page 128).

***Example 39. 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 39 on page 128 can be explained as follows:

❶ A Artix ESB 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* on page 115.

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[1] and Web Services Policy 1.5—Attachment[2] specifications.

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

1. Add the policy feature to your client and server endpoint. Example 40 on page 130 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 40. 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/address
ing/metadata">
    <wsam:Addressing>
      <wsp:Policy>
        <wsam:NonAnonymousResponses/>
      </wsp:Policy>
    </wsam:Addressing>
</wsp:Policy>
```

2. Add a reliable messaging policy to the `wsdl:service` element—or any other WSDL element that can be used as an attachment point for policy or policy reference elements—to your WSDL file, as shown in Example 41 on page 130.

***Example 41. 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] http://www.w3.org/TR/2006/WD-ws-policy-20061117/
[2] http://www.w3.org/TR/2006/WD-ws-policy-attach-20061117/

```
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 ESB-specific attributes that are defined in the Artix ESB 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 ESB-Specific WS-RM Attributes

## Overview

To configure the Artix ESB-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 42 on page 133 shows a simple example.

***Example 42. Configuring Artix ESB-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 15 on page 133 shows the child elements of the `rmManager` Spring bean, defined in the http://cxf.apache.org/ws/rm/manager namespace.

***Table 15. 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 |

| Element | Description |
| --- | --- |
| `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 on page 141.

# Configuring Standard WS-RM Policy Attributes

**Overview**

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**

Table 16 on page 135 shows the elements defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace:

***Table 16. 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).<br><br>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 |

| Name | Description |
|---|---|
| | 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 a Artix ESB `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 ESB-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 43 on page 136 shows:

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

***Example 43. 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 44 on page 137.

***Example 44. 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://schem
as.xmlsoap.org/ws/2005/02/rm/policy">
                     <wsrm:AcknowledgementInterval Milliseconds="200" />
                   </wsrm:RMAssertion>
                   <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/address
ing/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 on page 130 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 45 on page 138 shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

***Example 45. Configuring WS-RM in an External Attachment***

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy" xmlns:wsa="ht
tp://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

## Overview

This subsection focuses on configuring WS-RM attributes from a use case point of view. Where an attribute is a standard WS-RM policy attribute, defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace, only the example of setting it in an `RMAssertion` within an `rmManager` Spring bean is shown. For details of how to set such attributes as a policy within a feature; in a WSDL file, or in an external attachment, see Configuring Standard WS-RM Policy Attributes on page 135.

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 46 on page 139 shows how to set the WS-RM base retransmission interval.

***Example 46. 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.

Example 47 on page 140 shows how to set the WS-RM exponential backoff for retransmission.

***Example 47. 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.

Example 48 on page 141 shows how to set the WS-RM acknowledgement interval.

***Example 48. 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 49 on page 141 shows how to set the WS-RM maximum unacknowledged messages threshold.

***Example 49. 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 50 on page 141 shows how to set the maximum length of an RM sequence.

***Example 50. 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 51 on page 142 shows how to set the WS-RM message delivery assurance.

***Example 51. 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

## Overview

The Artix ESB 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 ESB enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Artix ESB 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 Developing Artix® Applications with JAX-WS.

### Important

WS-RM persistence is supported for oneway calls only, and it is disabled by default.

## How it works

Artix ESB 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 ESB.

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

```
<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 ESB supports the properties shown in Table 17 on page 144.

***Table 17. 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 52 on page 144 enables the JDBC-based store that comes with Artix ESB, while setting the driverClassName and url to non-default values.

***Example 52. 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 ESB runtime.*

# Introduction to High Availability

## Overview

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 ESB delivers the requests to one of the member replicas. The routing to a replica is transparent to the client.

## HA with static failover

Artix ESB 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[1].

[1] ../../locator_guide/index.htm

# Enabling HA with Static Failover

**Overview**

To enable HA with static failover, you must:

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

**Encode replica details in your service WSDL file**

You must encode the details of the replicas in your cluster in your service WSDL file. Example 53 on page 148 shows a WSDL file extract that defines a service cluster of three replicas.

***Example 53. 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 53 on page 148 can be explained as follows:

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

1. `Replica1`
2. `Replica2`
3. `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 54 on page 149

***Example 54. Enabling HA with Static Failover—Client Configuration***

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

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

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

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

</beans>
```

# Configuring HA with Static Failover

**Overview**

By default, HA with static failover uses a sequential strategy when selecting a replica service if the original service with which a client is communicating becomes unavailable or fails. The sequential strategy selects a replica service in the same sequential order every time it is used. Selection is determined by Artix ESB'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 55 on page 150 to your client configuration file:

***Example 55. 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 55 on page 150 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

**Overview**

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

1. Configure your service to register with the Artix locator on page 151
2. Configure your client to use locator meditated failover on page 152
3. Ensure the Artix locator is running on page 153

**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 56 on page 151 to your server configuration file.

***Example 56. 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 56 on page 151 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 57 on page 152 to your client configuration file.

***Example 57. Configuring your Client to Use Locator Mediated Failover***

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

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 57 on page 152 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[2].

[2] ../../locator_guide/index.htm

# Configuring HA with Dynamic Failover

## Overview

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



## Enabling Artix locator to check the state of a registered service

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.locat
or.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.locat
or.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.locat
or.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

## Overview

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

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

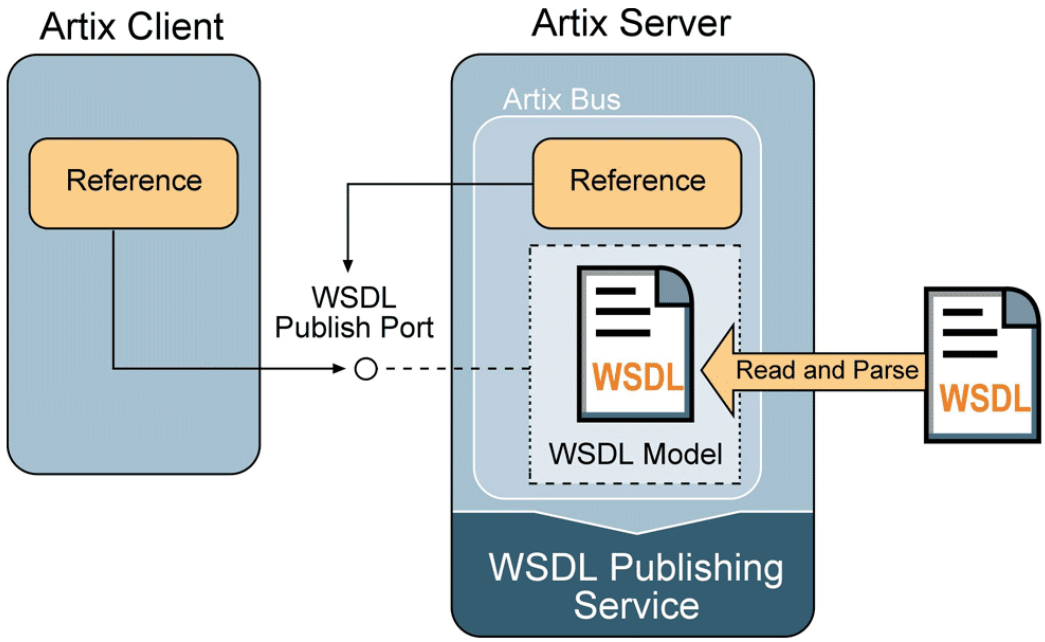
It is recommended that you use the WSDL publishing service for any applications that generate and export references. To use references, the consumer must have access to the WSDL file referred to by the reference. The simplest way to accomplish this is to use the WSDL publishing service.

Figure 5 on page 159 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 on page 165). If a client accesses the WSDL location URL, the server converts the WSDL model to XML on the fly and returns the WSDL contract in a HTTP message.

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




**Multiple transports**

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

# Configuring the WSDL Publishing Service

**Overview**

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

## Note

If you want to run the WSDL publishing service in a servlet container, please refer to Configuring for Use in a Servlet Container on page 163.

**Configuration file**

Example 58 on page 160 shows an example of such a configuration file. It is written using plain Spring beans. For more detailed information on each of the configuration options, see WSDL publishing service configuration options on page 161:

***Example 58. 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="ht
tp://cxf.apache.org/ http://schemas.iona.com/"/>

</beans>
```

The configuration shown in Example 58 on page 160 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 18 on page 161 describes each of the WSDL publishing service configuration options.

***Table 18. 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`. |

| Configuration Option | Description |
|---|---|
| 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`. |
| processWSDL | A string that specifies the type of processing that is done on the WSDL file before the WSDL file is published.<br><br>The processWSDL option has three possible values:<br><br>• `none`—no processing of the WSDL file takes place; that is, the WSDL document is published as is.<br><br>• `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.<br><br>• `standard`—same as `artix`, but non-standard extensions are also removed.<br><br>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

**Overview**

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* on page 97.

**Configuration steps**

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

1. Create a spring.xml configuration file on page 163
2. Configure a listener class in the web.xml file on page 164

**Create a spring.xml configuration file**

Create a `spring.xml` configuration file as shown in Example 59 on page 163 and include it in the `WEB-INF` directory of your application WAR file.

***Example 59. Configuring Artix WSDL Publish Service for Deployment to a Servlet Container***

```
<beans 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://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 60 on page 164 to your application's `web.xml` file. Include the `web.xml` file in the `WEB-INF` directory of your application WAR file.

***Example 60. 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

## Overview

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

The WSDL publishing service supports the following queries:

`?wsdl`

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:

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

- Using `?services`:

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

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.

# Accessing Services Using UDDI

*Artix provides support for Universal Description, Discovery and Integration (UDDI).*

# Introduction to UDDI

**Overview**

A Universal Description, Discovery and Integration (UDDI) registry is a form of database that facilitates the storage and retrieval of Web services endpoints. It is particularly useful for making Web services available on the Internet. Instead of making your service WSDL contract available to clients in the form of a file, you can publish the WSDL contract in a UDDI registry. Clients can then query the UDDI registry and retrieve the WSDL contract at runtime.

**Sample applications**

Artix includes UDDI sample applications, which can be found in the following directories:

- *ArtixInstallDir*`/java/samples/integration/uddi/client`
- *ArtixInstallDir*`/java/samples/integration/uddi/juddi`

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

**jUDDI**

Artix includes an open source UDDI registry called jUDDI. The sample applications use this registry to store UDDI information. For more information, see http://ws.apache.org/juddi/.

# Configuring a Client to Use UDDI

**Overview**

Clients can be configured to dynamically retrieve service WSDL contracts from a UDDI registry without the need for UDDI-specific code.

**Client code**

The following client code is valid for use with a UDDI registry once the client is configured to use UDDI (see Client configuration on page 169).

***Example 61. Programming an Application to Use a UDDI Registry***

```
QName serviceQName = new QName("http://hello", "HelloService");
HelloService service = new HelloService(serviceQName, null);
```

**Client configuration**

To configure a JAX-WS client to use UDDI add the configuration shown in Example 62 on page 169 to the client's configuration file:

***Example 62. UDDI Client Configuration***

```
<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.springframe
work.org/schema/beans/spring-beans.xsd">

❶     <bean id="UddiClientSupport" class="com.iona.cxf.uddi.client.UddiResolver">
        <property name="bus" ref="cxf"/>
❷         <property name="uddiUrl" value="http://localhost:8888/uddi/inquire"/>
    </bean>
</beans>
```

The configuration shown in Example 62 on page 169 can be explained as follows:

❶ Specifies the UDDI resolver, which is used to query the UDDI registry when the client requests a service endpoint. The client code does not have to explicitly specify UDDI—the UDDI resolver plugs in at the bus level and queries the UDDI registry.

❷ Specifies the inquire URL for the UDDI repository. In the example shown, the inquire URL specifies the jUDDI repository that ships with Artix.

# Appendix A. Artix ESB 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 |

# Index

## L



## M



## P



## R



## S



## U



## W