Fuse ESB

Configuring Web Service Endpoints

Version 4.4.1
Sept. 2011

Configuring Web Service Endpoints

Version 4.4.1

Updated: 06 Jun 2013

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Configuring JAX-WS Endpoints

JAX-WS 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

Fuse Services Framework 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 1.1 on page 16 describes the attribute of the `jaxws:endpoint` element.

Table 1.1. 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.
`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 137.

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 Fuse Services Framework APIs, such as `Endpoint.publish()` or `Service.getPort()`. The default is `false`. Setting this to `true` does the following: • Changes the internal name of the bean by appending `.jaxws-endpoint` to its id • Makes the bean abstract
`publishedEndpointUrl`	The URL that is placed in the `address` element of the generated WSDL. If this value is not specified, the value of the `address` attribute is used. This attribute is useful when the "public" URL is not be the same as the URL on which the service is deployed.

In addition to the attributes listed in Table 1.1 on page 16, you might need to use multiple `xmlns:`*`shortName`* attributes to declare the namespaces used by the `endpointName` and `serviceName` attributes.

Example

Example 1.1 on page 18 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 1.1. 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 1.2 on page 18 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 1.2. 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 Fuse Services Framework specific object. If you are using a pure Spring approach to building your services, you will not be forced to use Fuse Services Framework 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 1.2 on page 20 describes the attribute of the `jaxws:server` element.

Table 1.2. 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.
`bus`	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful

Attribute	Description
	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 137.
`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 Fuse Services Framework APIs, such as `Endpoint.publish()` or `Service.getPort()`. The default is `false`. Setting this to `true` does the following: • Changes the internal name of the bean by appending `.jaxws-endpoint` to its id • Makes the bean abstract

In addition to the attributes listed in Table 1.2 on page 20, you might need to use multiple `xmlns:`*`shortName`* attributes to declare the namespaces used by the `endpointName` and `serviceName` attributes.

Example

Example 1.3 on page 22 shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published.

Example 1.3. 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 1.3 on page 23 describes the child elements that `jaxws:endpoint` supports.

Table 1.3. Elements Used to Configure JAX-WS Service Providers

Element	Description
`jaxws:handlers`	Specifies a list of JAX-WS `Handler` implementations for processing messages.
`jaxws:inInterceptors`	Specifies a list of interceptors that process inbound requests. For more information see .
`jaxws:inFaultInterceptors`	Specifies a list of interceptors that process inbound fault messages. For more information see .
`jaxws:outInterceptors`	Specifies a list of interceptors that process outbound replies. For more information see .
`jaxws:outFaultInterceptors`	Specifies a list of interceptors that process outbound fault messages. For more information see .
`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]

Element	Description
`jaxws:dataBinding` [b]	Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition.
`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 Fuse Services Framework. 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 1.4 on page 25 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 1.4. 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 137.
`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.

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

In addition to the attributes listed in Table 1.4 on page 25, 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 1.5 on page 27 describes the child element's you can use to configure a JAX-WS consumer.

Table 1.5. 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 Fuse Services Framework. 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.
`jaxws:inInterceptors`	Specifies a list of interceptors that process inbound responses. For more information see *Developing Apache CXF Interceptors*.
`jaxws:inFaultInterceptors`	Specifies a list of interceptors that process inbound fault messages. For more information see *Developing Apache CXF Interceptors*.
`jaxws:outInterceptors`	Specifies a list of interceptors that process outbound requests. For more information see *Developing Apache CXF Interceptors*.

Element	Description
`jaxws:outFaultInterceptors`	Specifies a list of interceptors that process outbound fault messages. For more information see *Developing Apache CXF Interceptors*.
`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 1.4 on page 28 shows a simple consumer configuration.

Example 1.4. 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>
```

Chapter 2. Configuring the HTTP Transport

The Fuse Service Framework HTTP transport is highly configurable.

Configuring a Consumer

HTTP consumer endpoints can specify a number of HTTP connection attributes including whether the endpoint automatically accepts redirect responses, whether the endpoint can use chunking, whether the endpoint will request a keep-alive, and how the endpoint interacts with proxies. In addition to the HTTP connection properties, an HTTP consumer endpoint can specify how it is secured.

A consumer endpoint can be configured using two mechanisms:

- Configuration
- WSDL

Using Configuration

Namespace

The elements used to configure an HTTP consumer endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the lines shown in Example 2.1 on page 31 to the `beans` element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 2.1. HTTP Consumer Configuration Namespace

```
<beans ...
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       ...
       xsi:schemaLocation="...
                           http://cxf.apache.org/transports/http/configuration
                              http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           ...">
```

The conduit element

You configure an HTTP consumer endpoint using the `http-conf:conduit` element and its children. The `http-conf:conduit` element takes a single attribute, `name`, that specifies the WSDL `port` element corresponding to the endpoint. The value for the `name` attribute takes the form *`portQName`*`.http-conduit`.
Example 2.2 on page 31 shows the `http-conf:conduit` element that would be used to add configuration for an endpoint that is specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> when the endpoint's target namespace is `http://widgets.widgetvendor.net`.

Example 2.2. `http-conf:conduit` Element

```
...
  <http-conf:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
    ...
  </http-conf:conduit>
...
```

The `http-conf:conduit` element has child elements that specify configuration information. They are described in Table 2.1 on page 32.

Table 2.1. Elements Used to Configure an HTTP Consumer Endpoint

Element	Description
`http-conf:client`	Specifies the HTTP connection properties such as timeouts, keep-alive requests, content types, etc. See "The `client` element" on page 32.
`http-conf:authorization`	Specifies the parameters for configuring the basic authentication method that the endpoint uses preemptively. The preferred approach is to supply a Basic Authentication Supplier object.
`http-conf:proxyAuthorization`	Specifies the parameters for configuring basic authentication against outgoing HTTP proxy servers.
`http-conf:tlsClientParameters`	Specifies the parameters used to configure SSL/TLS.
`http-conf:basicAuthSupplier`	Specifies the bean reference or class name of the object that supplies the basic authentication information used by the endpoint, either preemptively or in response to a `401` HTTP challenge.
`http-conf:trustDecider`	Specifies the bean reference or class name of the object that checks the HTTP(S) `URLConnection` object to establish trust for a connection with an HTTPS service provider before any information is transmitted.

The client element

The `http-conf:client` element is used to configure the non-security properties of a consumer endpoint's HTTP connection. Its attributes, described in Table 2.2 on page 32, specify the connection's properties.

Table 2.2. HTTP Consumer Configuration Attributes

Attribute	Description
`ConnectionTimeout`	Specifies the amount of time, in milliseconds, that the consumer attempts to establish a connection before it times out. The default is `30000`. `0` specifies that the consumer will continue to send the request indefinitely.

Attribute	Description
`ReceiveTimeout`	Specifies the amount of time, in milliseconds, that the consumer will wait for a response before it times out. The default is `30000`. `0` specifies that the consumer will wait indefinitely.
`AutoRedirect`	Specifies if the consumer will automatically follow a server issued redirection. The default is `false`.
`MaxRetransmits`	Specifies the maximum number of times a consumer will retransmit a request to satisfy a redirect. The default is `-1` which specifies that unlimited retransmissions are allowed.
`AllowChunking`	Specifies whether the consumer will send requests using chunking. The default is `true` which specifies that the consumer will use chunking when sending requests. Chunking cannot be used if either of the following are true: • `http-conf:basicAuthSupplier` is configured to provide credentials preemptively. • `AutoRedirect` is set to `true`. In both cases the value of `AllowChunking` is ignored and chunking is disallowed.
`Accept`	Specifies what media types the consumer is prepared to handle. The value is used as the value of the HTTP `Accept` property. The value of the attribute is specified using multipurpose internet mail extensions (MIME) types.
`AcceptLanguage`	Specifies what language (for example, American English) the consumer prefers for the purpose of receiving a response. The value is used as the value of the HTTP `AcceptLanguage` property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined

Attribute	Description
	by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.
`AcceptEncoding`	Specifies what content encodings the consumer is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP `AcceptEncoding` property.
`ContentType`	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP `ContentType` property. The default is `text/xml`. For web services, this should be set to `text/xml`. If the client is sending HTML form data to a CGI script, this should be set to `application/x-www-form-urlencoded`. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to `application/octet-stream`.
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP `Host` property. This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).
`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • `Keep-Alive` — Specifies that the consumer wants the connection kept open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it.

Attribute	Description
	• `close`(default) — Specifies that the connection to the server is closed after each request/response sequence.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a consumer to a service provider. See "Consumer Cache Control Directives" on page 39.
`Cookie`	Specifies a static cookie to be sent with all requests.
`BrowserType`	Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based on the client that is sending the request.
`Referer`	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP `Referer` property. This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. If the `AutoRedirect` attribute is set to `true` and the request is redirected, any value specified in the `Referer` attribute is overridden. The value of the HTTP Referer property is set to the URL of the service that redirected the consumer's original request.
`DecoupledEndpoint`	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate provider->consumer connection. For more information on using decoupled endpoints see, "Using the HTTP Transport in Decoupled Mode" on page 48.

Attribute	Description
	You must configure both the consumer endpoint and the service provider endpoint to use WS-Addressing for the decoupled endpoint to work.
`ProxyServer`	Specifies the URL of the proxy server through which requests are routed.
`ProxyServerPort`	Specifies the port number of the proxy server through which requests are routed.
`ProxyServerType`	Specifies the type of proxy server used to route requests. Valid values are: • `HTTP`(default) • `SOCKS`

Example

Example 2.3 on page 36 shows the configuration of an HTTP consumer endpoint that wants to keep its connection to the provider open between requests, that will only retransmit requests once per invocation, and that cannot use chunking streams.

Example 2.3. HTTP Consumer Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">


  <http-conf:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">

    <http-conf:client Connection="Keep-Alive"
                      MaxRetransmits="1"
                      AllowChunking="false" />
  </http-conf:conduit>
</beans>
```

More information

For more information on HTTP conduits see Appendix C on page 153.

Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP consumer endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the line shown in Example 2.4 on page 38 to the `definitions` element of your endpoint's WSDL document.

Example 2.4. HTTP Consumer WSDL Element's Namespace

```
<definitions ...
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

The client element

The `http-conf:client` element is used to specify the connection properties of an HTTP consumer in a WSDL document. The `http-conf:client` element is a child of the WSDL `port` element. It has the same attributes as the `client` element used in the configuration file. The attributes are described in Table 2.2 on page 32.

Example

Example 2.5 on page 38 shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 2.5. WSDL to Configure an HTTP Consumer Endpoint

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

Consumer Cache Control Directives

Table 2.3 on page 39 lists the cache control directives supported by an HTTP consumer.

Table 2.3. `http-conf:client` Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store either any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that is still fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

Configuring a Service Provider

HTTP service provider endpoints can specify a number of HTTP connection attributes including if it will honor keep alive requests, how it interacts with caches, and how tolerant it is of errors in communicating with a consumer.

A service provider endpoint can be configured using two mechanisms:

- Configuration
- WSDL

Using Configuration

Namespace

The elements used to configure an HTTP provider endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the lines shown in Example 2.6 on page 41 to the `beans` element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 2.6. HTTP Provider Configuration Namespace

```
<beans ...
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       ...
       xsi:schemaLocation="...
                           http://cxf.apache.org/transports/http/configuration
                              http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           ...">
```

The destination element

You configure an HTTP service provider endpoint using the `http-conf:destination` element and its children. The `http-conf:destination` element takes a single attribute, `name`, that specifies the WSDL `port` element that corresponds to the endpoint. The value for the `name` attribute takes the form *`portQName`*`.http-destination`. Example 2.7 on page 41 shows the `http-conf:destination` element that is used to add configuration for an endpoint that is specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> when the endpoint's target namespace is `http://widgets.widgetvendor.net`.

Example 2.7. `http-conf:destination` Element

```
...
  <http-conf:destination name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-destin
ation">
    ...
  </http-conf:destination>
...
```

The `http-conf:destination` element has a number of child elements that specify configuration information. They are described in Table 2.4 on page 42.

Table 2.4. Elements Used to Configure an HTTP Service Provider Endpoint

Element	Description
`http-conf:server`	Specifies the HTTP connection properties. See "The `server` element" on page 42.
`http-conf:contextMatchStrategy`	Specifies the parameters that configure the context match strategy for processing HTTP requests.
`http-conf:fixedParameterOrder`	Specifies whether the parameter order of an HTTP request handled by this destination is fixed.

The server element

The `http-conf:server` element is used to configure the properties of a service provider endpoint's HTTP connection. Its attributes, described in Table 2.5 on page 42, specify the connection's properties.

Table 2.5. HTTP Service Provider Configuration Attributes

Attribute	Description
`ReceiveTimeout`	Sets the length of time, in milliseconds, the service provider attempts to receive a request before the connection times out. The default is `30000`. `0` specifies that the provider will not timeout.
`SuppressClientSendErrors`	Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. The default is `false`; exceptions are thrown on encountering errors.
`SuppressClientReceiveErrors`	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a consumer. The default is `false`; exceptions are thrown on encountering errors.
`HonorKeepAlive`	Specifies whether the service provider honors requests for a connection to remain open after a response has been sent. The default is `false`; keep-alive requests are ignored.
`RedirectURL`	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is

Attribute	Description
	set to `302` and the status description is set to `Object Moved`. The value is used as the value of the HTTP RedirectURL property.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a service provider to a consumer. See "Service Provider Cache Control Directives" on page 46.
`ContentLocation`	Sets the URL where the resource being sent in a response is located.
`ContentType`	Specifies the media type of the information being sent in a response. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType location.
`ContentEncoding`	Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`. This value is used as the value of the HTTP ContentEncoding property. The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Fuse Services Framework performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.
`ServerType`	Specifies what type of server is sending the response. Values take the form *`program-name/version`*; for example, `Apache/1.2.5`.

Example

Example 2.8 on page 44 shows the configuration for an HTTP service provider endpoint that honors keep-alive requests and suppresses all communication errors.

Example 2.8. HTTP Service Provider Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">


  <http-conf:destination name="{http://apache.org/hello_world_soap_http}SoapPort.http-des
tination">
    <http-conf:server SuppressClientSendErrors="true"
                      SuppressClientReceiveErrors="true"
                      HonorKeepAlive="true" />
  </http-conf:destination>
</beans>
```

Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP provider endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. To use the HTTP configuration elements you must add the line shown in Example 2.9 on page 45 to the `definitions` element of your endpoint's WSDL document.

Example 2.9. HTTP Provider WSDL Element's Namespace

```
<definitions ...
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

The server element

The `http-conf:server` element is used to specify the connection properties of an HTTP service provider in a WSDL document. The `http-conf:server` element is a child of the WSDL `port` element. It has the same attributes as the `server` element used in the configuration file. The attributes are described in Table 2.5 on page 42.

Example

Example 2.10 on page 45 shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 2.10. WSDL to Configure an HTTP Service Provider Endpoint

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:server CacheControl="no-cache" />
  </port>
</service>
```

Service Provider Cache Control Directives

Table 2.6 on page 46 lists the cache control directives supported by an HTTP service provider.

Table 2.6. `http-conf:server` Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
public	Any cache can store the response.
private	Public (*shared*) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of the response or any part of the request that invoked it.
no-transform	Caches must not modify the media type or location of the content in a response between a server and a client.
must-revalidate	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.
proxy-revalidate	Does the same as must-revalidate, except that it can only be enforced on shared caches and is ignored by private unshared caches. When using this directive, the public cache directive must also be used.
max-age	Clients can accept a response whose age is no greater that the specified number of seconds.
s-max-age	Does the same as max-age, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by s-max-age overrides the age specified by max-age. When using

Directive	Behavior
	this directive, the proxy-revalidate directive must also be used.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

Using the HTTP Transport in Decoupled Mode

Overview

In normal HTTP request/response scenarios, the request and the response are sent using the same HTTP connection. The service provider processes the request and responds with a response containing the appropriate HTTP status code and the contents of the response. In the case of a successful request, the HTTP status code is set to `200`.

In some instances, such as when using WS-RM or when requests take an extended period of time to execute, it makes sense to decouple the request and response message. In this case the service providers sends the consumer a `202 Accepted` response to the consumer over the back-channel of the HTTP connection on which the request was received. It then processes the request and sends the response back to the consumer using a new decoupled server->client HTTP connection. The consumer runtime receives the incoming response and correlates it with the appropriate request before returning to the application code.

Configuring decoupled interactions

Using the HTTP transport in decoupled mode requires that you do the following:

1. Configure the consumer to use WS-Addressing.

   See "Configuring an endpoint to use WS-Addressing" on page 48.

2. Configure the consumer to use a decoupled endpoint.

   See "Configuring the consumer" on page 49.

3. Configure any service providers that the consumer interacts with to use WS-Addressing.

   See "Configuring an endpoint to use WS-Addressing" on page 48.

Configuring an endpoint to use WS-Addressing

Specify that the consumer and any service provider with which the consumer interacts use WS-Addressing.

You can specify that an endpoint uses WS-Addressing in one of two ways:

- Adding the `wswa:UsingAddressing` element to the endpoint's WSDL `port` element as shown in Example 2.11 on page 49.

Example 2.11. Activating WS-Addressing using WSDL

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wswa:UsingAddressing xmlns:wswa="http://www.w3.org/2005/02/addressing/wsdl"/>
  </port>
</service>
...
```

- Adding the WS-Addressing policy to the endpoint's WSDL `port` element as shown in Example 2.12 on page 49.

Example 2.12. Activating WS-Addressing using a Policy

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy">
      <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
  </port>
</service>
...
```

Note

The WS-Addressing policy supersedes the `wswa:UsingAddressing` WSDL element.

Configuring the consumer

Configure the consumer endpoint to use a decoupled endpoint using the `DecoupledEndpoint` attribute of the `http-conf:conduit` element.

Example 2.13 on page 50 shows the configuration for setting up the endpoint defined in Example 2.11 on page 49 to use use a decoupled endpoint. The consumer now receives all responses at `http://widgetvendor.net/widgetSellerInbox`.

Example 2.13. Configuring a Consumer to Use a Decoupled HTTP Endpoint

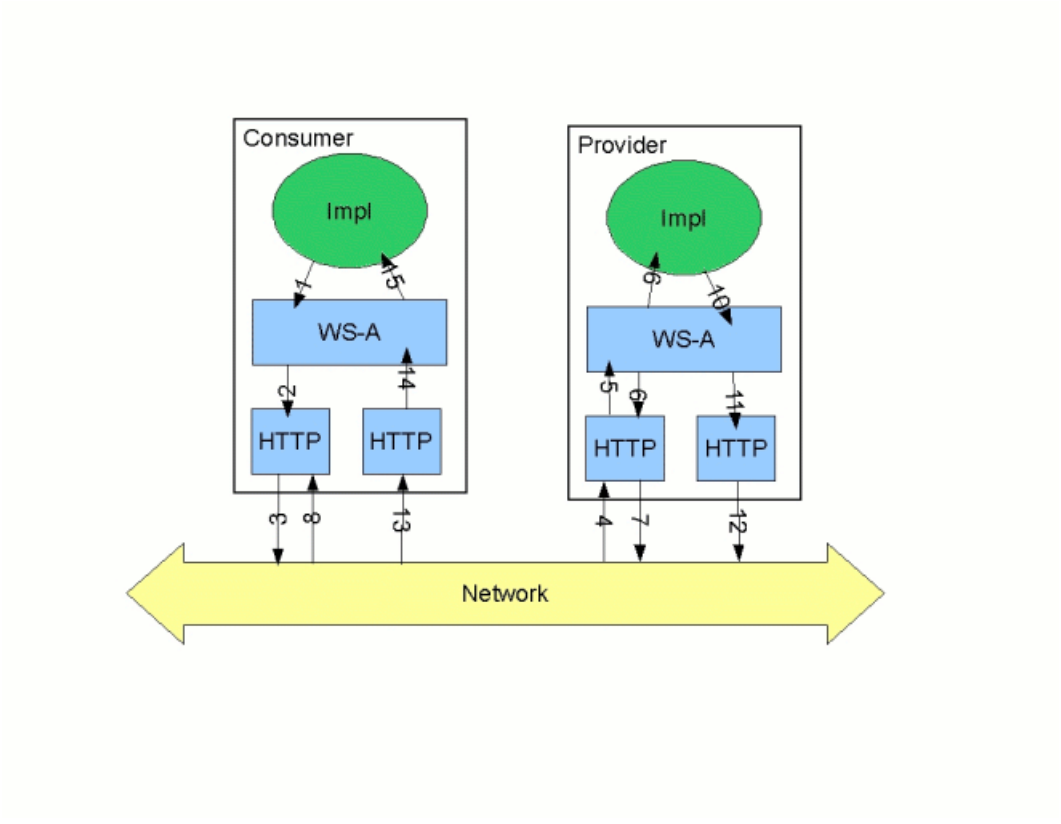
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">


  <http:conduit name="{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
    <http:client DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

How messages are processed

Using the HTTP transport in decoupled mode adds extra layers of complexity to the processing of HTTP messages. While the added complexity is transparent to the implementation level code in an application, it might be important to understand what happens for debugging reasons.

Figure 2.1 on page 51 shows the flow of messages when using HTTP in decoupled mode.

Figure 2.1. Message Flow in for a Decoupled HTTP Transport



A request starts the following process:

1. The consumer implementation invokes an operation and a request message is generated.
2. The WS-Addressing layer adds the WS-A headers to the message.

   When a decoupled endpoint is specified in the consumer's configuration, the address of the decoupled endpoint is placed in the WS-A ReplyTo header.
3. The message is sent to the service provider.
4. The service provider receives the message.
5. The request message from the consumer is dispatched to the provider's WS-A layer.

6. Because the WS-A ReplyTo header is not set to anonymous, the provider sends back a message with the HTTP status code set to `202`, acknowledging that the request has been received.
7. The HTTP layer sends a `202 Accepted` message back to the consumer using the original connection's back-channel.
8. The consumer receives the `202 Accepted` reply on the back-channel of the HTTP connection used to send the original message.

   When the consumer receives the `202 Accepted` reply, the HTTP connection closes.
9. The request is passed to the service provider's implementation where the request is processed.
10. When the response is ready, it is dispatched to the WS-A layer.
11. The WS-A layer adds the WS-Addressing headers to the response message.
12. The HTTP transport sends the response to the consumer's decoupled endpoint.
13. The consumer's decoupled endpoint receives the response from the service provider.
14. The response is dispatched to the consumer's WS-A layer where it is correlated to the proper request using the WS-A RelatesTo header.
15. The correlated response is returned to the client implementation and the invoking call is unblocked.

Chapter 3. Using SOAP Over JMS

Fuse Services Framework implements the W3C standard SOAP/JMS transport. This standard is intended to provide a more robust alternative to SOAP/HTTP services. Fuse Services Framework applications using this transport should be able to interoperate with applications that also implement the SOAP/JMS standard. The transport is configured directly in an endpoint's WSDL.

Basic configuration

Overview

The SOAP over JMS protocol[1] is defined by the World Wide Web Consortium(W3C) as a way of providing a more reliable transport layer to the customary SOAP/HTTP protocol used by most services. The Fuse Services Framework implementation is fully compliant with the specification and should be compatible with any framework that is also compliant.

This transport uses JNDI to find the JMS destinations. When an operation is invoked, the request is packaged as a SOAP message and sent in the body of a JMS message to the specified destination.

To use the SOAP/JMS transport:

1. Specify that the transport type is SOAP/JMS.
2. Specify the target destination using a JMS URI.
3. Optionally, configure the JNDI connection.
4. Optionally, add additional JMS configuration.

Specifying the JMS transport type

You configure a SOAP binding to use the JMS transport when specifying the WSDL binding. You set the `soap:binding` element's `transport` attribute to `http://www.w3.org/2010/soapjms/`. Example 3.1 on page 54 shows a WSDL binding that uses SOAP/JMS.

Example 3.1. SOAP over JMS binding specification

```
<wsdl:binding ... >
  <soap:binding style="document"
                transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

Specifying the target destination

You specify the address of the JMS target destination when specifying the WSDL port for the endpoint. The address specification for a SOAP/JMS endpoint uses the same `soap:address` element and attribute as a

[1] http://www.w3.org/TR/soapjms/

SOAP/HTTP endpoint. The difference is the address specification. JMS endpoints use a JMS URI as defined in the URI Scheme for JMS 1.0[2]. Example 3.2 on page 55 shows the syntax for a JMS URI.

Example 3.2. JMS URI syntax

```
jms:variant:destination?options
```

Table 3.1 on page 55 describes the available variants for the JMS URI.

Table 3.1. JMS URI variants

Variant	Description
jndi	Specifies that the destination is a JNDI name for the target destination. When using this variant, you must provide the configuration for accessing the JNDI provider.
topic	Specifies that the destination is the name of the topic to be used as the target destination. The string provided is passed into `Session.createTopic()` to create a representation of the destination.
queue	Specifies that the destination is the name of the queue to be used as the target destination. The string provided is passed into `Session.createQueue()` to create a representation of the destination.

The *`options`* portion of a JMS URI are used to configure the transport and are discussed in "JMS URIs" on page 57.

Example 3.3 on page 55 shows the WSDL port entry for a SOAP/JMS endpoint whose target destination is looked up using JNDI.

Example 3.3. SOAP/JMS endpoint address

```
<wsdl:port ... >
  ...
  <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

For working with SOAP/JMS services in Java see "Using SOAP over JMS" in *Developing Applications Using JAX-WS*.

[2] http://tools.ietf.org/id/draft-merrick-jms-uri-06.txt

Configuring JNDI and the JMS transport

The SOAP/JMS provides several ways to configure the JNDI connection and the JMS transport:

- Using the JMS URI
- Using WSDL extensions

JMS URIs

Overview

When using SOAP/JMS, a JMS URI is used to specify the endpoint's target destination. The JMS URI can also be used to configure JMS connection by appending one or more options to the URI. These options are detailed in the IETF standard, URI Scheme for Java Message Service 1.0[3]. They can be used to configure the JNDI system, the reply destination, the delivery mode to use, and other JMS properties.

Syntax

As shown in Example 3.2 on page 55, you can append one or more options to the end of a JMS URI by separating them from the destination's address with a question mark(?). Multiple options are separated by an ampersand(&). Example 3.4 on page 57 shows the syntax for using multiple options in a JMS URI.

Example 3.4. Syntax for JMS URI options

```
jmsAddress?option1=value1&option2=value2&...optionN=valueN
```

JMS properties

Table 3.2 on page 57 shows the URI options that affect the JMS transport layer.

Table 3.2. JMS properties settable as URI options

Property	Default	Description
`deliveryMode`	`PERSISTENT`	Specifies whether to use JMS `PERSISTENT` or `NON_PERSISTENT` message semantics. In the case of `PERSISTENT` delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas `NON_PERSISTENT` messages are kept in memory only.
`replyToName`		Explicitly specifies the reply destination to appear in the `JMSReplyTo` header. Setting this property is recommended for applications that have request-reply

[3] http://tools.ietf.org/id/draft-merrick-jms-uri-06.txt

Property	Default	Description
		semantics because the JMS provider will assign a temporary reply queue if one is not explicitly set. The value of this property has an interpretation that depends on the variant specified in the JMS URI: • `jndi` variant—the JNDI name of the destination • `queue` or `topic` variants—the actual name of the destination
`priority`	`4`	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`timeToLive`	`0`	Time (in milliseconds) after which the message will be discarded by the JMS provider. A value of `0` represents an infinite lifetime (the default).

JNDI properties

Table 3.3 on page 58 shows the URI options that can be used to configure JNDI for this endpoint.

Table 3.3. JNDI properties settable as URI options

Property	Description
`jndiConnectionFactoryName`	Specifies the JNDI name of the JMS connection factory.
`jndiInitialContextFactory`	Specifies the fully qualified Java class name of the JNDI provider (which must be of `javax.jms.InitialContextFactory` type). Equivalent to setting the `java.naming.factory.initial` Java system property.

Property	Description
`jndiURL`	Specifies the URL that initializes the JNDI provider. Equivalent to setting the `java.naming.provider.url` Java system property.

Additional JNDI properties

The properties, `java.naming.factory.initial` and `java.naming.provider.url`, are standard properties, which are required to initialize any JNDI provider. Sometimes, however, a JNDI provider might support custom properties in addition to the standard ones. In this case, you can set an arbitrary JNDI property by setting a URI option of the form `jndi-`*`PropertyName`*.

For example, if you were using SUN's LDAP implementation of JNDI, you could set the JNDI property, `java.naming.factory.control`, in a JMS URI as shown in Example 3.5 on page 59.

Example 3.5. Setting a JNDI property in a JMS URI

```
jms:queue:FOO.BAR?jndi-java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
```

Example

If the JMS provider is *not* already configured, it is possible to provide the requisite JNDI configuration details in the URI using options (see Table 3.3 on page 58). For example, to configure an endpoint to use the Apache ActiveMQ JMS provider and connect to the queue called `test.cxf.jmstransport.queue`, use the URI shown in Example 3.6 on page 59.

Example 3.6. JMS URI that configures a JNDI connection

```
jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
?jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContextFactory
&jndiConnectionFactoryName=ConnectionFactory
&jndiURL=tcp://localhost:61616
```

WSDL extensions

Overview

You can specify the basic configuration of the JMS transport by inserting WSDL extension elements into the contract, either at binding scope, service scope, or port scope. The WSDL extensions enable you to specify the properties for bootstrapping a JNDI `InitialContext`, which can then be used to look up JMS destinations. You can also set some properties that affect the behavior of the JMS transport layer.

SOAP/JMS namespace

the SOAP/JMS WSDL extensions are defined in the http://www.w3.org/2010/soapjms/ namespace. To use them in your WSDL contracts add the following setting to the `wsdl:definitions` element:

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

WSDL extension elements

Table 3.4 on page 60 shows all of the WSDL extension elements you can use to configure the JMS transport.

Table 3.4. SOAP/JMS WSDL extension elements

Element	Default	Description
`soapjms:jndiInitialContextFactory`		Specifies the fully qualified Java class name of the JNDI provider. Equivalent to setting the `java.naming.factory.initial` Java system property.
`soapjms:jndiURL`		Specifies the URL that initializes the JNDI provider. Equivalent to setting the `java.naming.provider.url` Java system property.
`soapjms:jndiContextParameter`		Enables you to specify an additional property for creating the JNDI `InitialContext`. Use the `name` and `value` attributes to specify the property.
`soapjms:jndiConnectionFactoryName`		Specifies the JNDI name of the JMS connection factory.

Element	Default	Description
`soapjms:deliveryMode`	`PERSISTENT`	Specifies whether to use JMS `PERSISTENT` or `NON_PERSISTENT` message semantics. In the case of `PERSISTENT` delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas `NON_PERSISTENT` messages are kept in memory only.
`soapjms:replyToName`		Explicitly specifies the reply destination to appear in the `JMSReplyTo` header. Setting this property is recommended for SOAP invocations that have request-reply semantics. If this property is not set the JMS provider allocates a temporary queue with an automatically generated name. The value of this property has an interpretation that depends on the variant specified in the JMS URI, as follows: • `jndi` variant—the JNDI name of the destination. • `queue` or `topic` variants—the actual name of the destination.
`soapjms:priority`	`4`	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`soapjms:timeToLive`	`0`	Time, in milliseconds, after which the message will be discarded by the JMS provider. A value of `0` represents an infinite lifetime.

Configuration scopes

The WSDL elements placement in the WSDL contract effect the scope of the configuration changes on the endpoints defined in the contract. The SOAP/JMS WSDL elements can be placed as children of either the `wsdl:binding` element, the `wsdl:service` element, or the `wsdl:port` element. The parent of the SOAP/JMS elements determine which of the following scopes the configuration is placed into.

Binding scope
: You can configure the JMS transport at the *binding scope* by placing extension elements inside the `wsdl:binding` element. Elements in this scope define the default configuration for all endpoints that use this binding. Any settings in the binding scope can be overridden at the service scope or the port scope.

Service scope
: You can configure the JMS transport at the *service scope* by placing extension elements inside a `wsdl:service` element. Elements in this scope define the default configuration for all endpoints in this service. Any settings in the service scope can be overridden at the port scope.

Port scope
: You can configure the JMS transport at the *port scope* by placing extension elements inside a `wsdl:port` element. Elements in the port scope define the configuration for this port. They override any defaults defined at the service scope or at the binding scope.

Example

Example 3.7 on page 62 shows a WSDL contract for a SOAP/JMS service. It configures the JNDI layer in the binding scope, the message delivery details in the service scope, and the reply destination in the port scope.

Example 3.7. WSDL contract with SOAP/JMS configuration

```
<wsd;definitions ...
❶    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
  ...
  <wsdl:binding name="JMSGreeterPortBinding" type="tns:JMSGreeterPortType">
    ...
❷    <soapjms:jndiInitialContextFactory>
      org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    ...
  </wsdl:binding>
```

```
  ...
  <wsdl:service name="JMSGreeterService">
    ...
❸    <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
    <soapjms:timeToLive>60000</soapjms:timeToLive>
    ...
    <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
❹       <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
❺       <soapjms:replyToName>
         dynamicQueues/greeterReply.queue
      </soapjms:replyToName>
      ...
    </wsdl:port>
    ...
  </wsdl:service>
  ...
</wsdl:definitions>
```

The WSDL in Example 3.7 on page 62 does the following:

❶ Declare the namespace for the SOAP/JMS extensions.
❷ Configure the JNDI connections in the binding scope.
❸ Configure the JMS delivery style to non-persistent and each message to live for one minute.
❹ Specify the target destination.
❺ Configure the JMS transport so that reply messages are delivered on the `greeterReply.queue` queue.

Chapter 4. Using Generic JMS

Fuse Services Framework provides a generic implementation of a JMS transport. The generic JMS transport is not restricted to using SOAP messages and allows for connecting to any application that uses JMS.

The Fuse Services Framework generic JMS transport can connect to any JMS provider and work with applications that exchange JMS messages with bodies of either `TextMessage` or `ByteMessage`.

There are two ways to enable and configure the JMS transport:

- JMS configuration bean
- WSDL

Using the JMS configuration bean

Overview

To simplify JMS configuration and make it more powerful, Fuse Services Framework uses a single JMS configuration bean to configure JMS endpoints. The bean is implemented by the `org.apache.cxf.transport.jms.JMSConfiguration` class. It can be used to either configure endpoint's directly or to configure the JMS conduits and destinations.

Specifying the configuration

You specify the JMS configuration by defining a bean of class `org.apache.cxf.transport.jms.JMSConfiguration`. The properties of the bean provide the configuration settings for the transport.

Table 4.1 on page 66 lists properties that are common to both providers and consumers.

Table 4.1. General JMS configuration properties

Property	Default	Description
`connectionFactory-ref`		Specifies a reference to a bean that defines a JMS `ConnectionFactory`.
`wrapInSingleConnectionFactory`	`true`	Specifies whether to wrap the `ConnectionFactory` with a Spring `SingleConnectionFactory`. Doing so can improve the performance of the JMS transport when the specified connection factory does not pool connections.
`reconnectOnException`	`false`	Specifies whether to create a new connection in the case of an exception. This property is only used when wrapping the connection factory with a Spring `SingleConnectionFactory`.
`targetDestination`		Specifies the JNDI name or provider specific name of a destination.
`replyDestination`		Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies.

Property	Default	Description
		For more details see "Using a Named Reply Destination" on page 80.
`destinationResolver`		Specifies a reference to a Spring `DestinationResolver`. This allows you to define how destination names are resolved. By default a `DynamicDestinationResolver` is used. It resolves destinations using the JMS providers features. If you reference a `JndiDestinationResolver` you can resolve the destination names using JNDI.
`transactionManager`		Specifies a reference to a Spring transaction manager. This allows the service to participate in JTA Transactions.
`taskExecutor`		Specifies a reference to a Spring `TaskExecutor`. This is used in listeners to decide how to handle incoming messages. By default the transport uses the Spring `SimpleAsyncTaskExecutor`.
`useJms11`	`false`	Specifies whether JMS 1.1 features are available.
`messageIdEnabled`	`true`	Specifies whether the JMS transport wants the JMS broker to provide message IDs. Setting this to `false` causes the endpoint to call its message producer's `setDisableMessageID()` method with a value of `true`. The JMS broker is then given a hint that it does not need to generate message IDs or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.

Property	Default	Description
messageTimestampEnabled	true	Specifies whether the JMS transport wants the JMS broker to provide message time stamps. Setting this to false causes the endpoint to call its message producer's setDisableMessageTimestamp() method with a value of true. The JMS broker is then given a hint that it does not need to generate time stamps or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.
cacheLevel	3	Specifies the level of caching allowed by the listener. Valid values are 0(CACHE_NONE), 1(CACHE_CONNECTION), 2(CACHE_SESSION), 3(CACHE_CONSUMER), 4(CACHE_AUTO).
pubSubNoLocal	false	Specifies whether to receive messages produced from the same connection.
receiveTimeout	0	Specifies, in milliseconds, the amount of time to wait for response messages. 0 means wait indefinitely.
explicitQosEnabled	false	Specifies whether the QoS settings like priority, persistence, and time to live are explicitly set for each message or if they are allowed to use default values.
deliveryMode	1	Specifies if a message is persistent. The two values are: • 1(NON_PERSISTENT)—messages will be kept memory • 2(PERSISTENT)—messages will be persisted to disk

Property	Default	Description
priority	4	Specifies the message's priority for the messages. JMS priority values can range from 0 to 9. The lowest priority is 0 and the highest priority is 9.
timeToLive	0	Specifies, in milliseconds, the message will be available after it is sent. 0 specifies an infinite time to live.
sessionTransacted	false	Specifies if JMS transactions are used.
concurrentConsumers	1	Specifies the minimum number of concurrent consumers created by the listener.
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers by listener.
maxConcurrentTasks	10	Specifies the maximum number of threads that handle the received requests.
messageSelector		Specifies the string value of the selector. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
subscriptionDurable	false	Specifies whether the server uses durrable subscriptions.
durableSubscriptionName		Specifies the string used to register the durable subscription.
messageType	text	Specifies how the message data will be packaged as a JMS message. `text` specifies that the data will be packaged as a `TextMessage`. `binary` specifies that the data will be packaged as an `ByteMessage`.
pubSubDomain	false	Specifies whether the target destination is a topic.

Property	Default	Description
`jmsProviderTibcoEms`	`false`	Specifies if your JMS provider is Tibco EMS. This causes the principal in the security context to be populated from the `JMS_TIBCO_SENDER` header.
`useMessageIDAsCorrelationID`	`false`	Specifies whether JMS will use the message ID to correlate messages. If not, the client will set a generated correlation ID.

As shown in Example 4.1 on page 70, the bean's properties are specified as attributes to the `bean` element. They are all declared in the Spring p namespace.

Example 4.1. JMS configuration bean

```
<bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
      p:connectionFactory-ref="connectionFactory"
      p:targetDestination="dynamicQueues/greeter.request.queue"
      p:pubSubDomain="false" />
```

Applying the configuration to an endpoint

The `JMSConfiguration` bean can be applied directly to both server and client endpoints using the Fuse Services Framework features mechanism. To do so:

1. Set the endpoint's `address` attribute to `jms://`.
2. Add a `jaxws:feature` element to the endpoint's configuration.
3. Add a bean of type `org.apache.cxf.transport.jms.JMSConfigFeature` to the feature.
4. Set the `bean` element's `p:jmsConfig-ref` attribute to the ID of the `JMSConfiguration` bean.

Example 4.2 on page 70 shows a JAX-WS client that uses the JMS configuration from Example 4.1 on page 70.

Example 4.2. Adding JMS configuration to a JAX-WS client

```
<jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"
              serviceClass="com.example.customerservice.CustomerService">
```

```
  <jaxws:features>
    <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
          p:jmsConfig-ref="jmsConfig"/>
  </jaxws:features>
</jaxws:client>
```

Applying the configuration to the transport

The `JMSConfiguration` bean can be applied to JMS conduits and JMS destinations using the `jms:jmsConfig-ref` element. The `jms:jmsConfig-ref` element's value is the ID of the `JMSConfiguration` bean.

Example 4.3 on page 71 shows a JMS conduit that uses the JMS configuration from Example 4.1 on page 70.

Example 4.3. Adding JMS configuration to a JMS conduit

```
<jms:conduit name="{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-con
duit">
  ...
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

Using WSDL to configure JMS

The WSDL extensions for defining a JMS endpoint are defined in the namespace `http://cxf.apache.org/transports/jms`. In order to use the JMS extensions you will need to add the line shown in Example 4.4 on page 72 to the definitions element of your contract.

Example 4.4. JMS WSDL extension namespace

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

Basic JMS configuration

Overview

The JMS address information is provided using the `jms:address` element and its child, the `jms:JMSNamingProperties` element. The `jms:address` element's attributes specify the information needed to identify the JMS broker and the destination. The `jms:JMSNamingProperties` element specifies the Java properties used to connect to the JNDI service.

Important

Information specified using the JMS feature will override the information in the endpoint's WSDL file.

Specifying the JMS address

The basic configuration for a JMS endpoint is done by using a `jms:address` element as the child of your service's `port` element. The `jms:address` element used in WSDL is identical to the one used in the configuration file. Its attributes are listed in Table 4.2 on page 73.

Table 4.2. JMS endpoint attributes

Attribute	Description
`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jmsDestinationName`	Specifies the JMS name of the JMS destination to which requests are sent.
`jmsReplyDestinationName`	Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see "Using a Named Reply Destination" on page 80.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which requests are sent.
`jndiReplyDestinationName`	Specifies the JNDI name bound to the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies.

Attribute	Description
	For more details see "Using a Named Reply Destination" on page 80.
`connectionUserName`	Specifies the user name to use when connecting to a JMS broker.
`connectionPassword`	Specifies the password to use when connecting to a JMS broker.

The `jms:address` WSDL element uses a `jms:JMSNamingProperties` child element to specify additional information needed to connect to a JNDI provider.

Specifying JNDI properties

To increase interoperability with JMS and JNDI providers, the `jms:address` element has a child element, `jms:JMSNamingProperties`, that allows you to specify the values used to populate the properties used when connecting to the JNDI provider. The `jms:JMSNamingProperties` element has two attributes: `name` and `value`. `name` specifies the name of the property to set. `value` attribute specifies the value for the specified property. `jms:JMSNamingProperties` element can also be used for specification of provider specific properties.

The following is a list of common JNDI properties that can be set:

1. `java.naming.factory.initial`
2. `java.naming.provider.url`
3. `java.naming.factory.object`
4. `java.naming.factory.state`
5. `java.naming.factory.url.pkgs`
6. `java.naming.dns.url`
7. `java.naming.authoritative`
8. `java.naming.batchsize`
9. `java.naming.referral`
10. `java.naming.security.protocol`
11. `java.naming.security.authentication`
12. `java.naming.security.principal`

13. `java.naming.security.credentials`

14. `java.naming.language`

15. `java.naming.applet`

For more details on what information to use in these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Example

Example 4.5 on page 75 shows an example of a JMS WSDL `port` specification.

Example 4.5. JMS WSDL port specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

JMS client configuration

Overview

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS `ByteMessage` or a JMS `TextMessage`.

When using an `ByteMessage` the consumer endpoint uses a byte[] as the method for storing data into and retrieving data from the JMS message body. When messages are sent, the message data, including any formating information, is packaged into a byte[] and placed into the message body before it is placed on the wire. When messages are received, the consumer endpoint will attempt to unmarshall the data stored in the message body as if it were packed in a byte[].

When using a `TextMessage`, the consumer endpoint uses a string as the method for storing and retrieving data from the message body. When messages are sent, the message information, including any format-specific information, is converted into a string and placed into the JMS message body. When messages are received the consumer endpoint will attempt to unmarshall the data stored in the JMS message body as if it were packed into a string.

When native JMS applications interact with Fuse Services Framework consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the Fuse Services Framework contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as `TextMessage`, the receiving JMS application will get a text message containing all of the SOAP envelope information.

Specifying the message type

The type of messages accepted by a JMS consumer endpoint is configured using the optional `jms:client` element. The `jms:client` element is a child of the WSDL `port` element and has one attribute:

Table 4.3. JMS Client WSDL Extensions

`messageType`	Specifies how the message data will be packaged as a JMS message. `text` specifies that the data will be packaged as a `TextMessage`. `binary` specifies that the data will be packaged as an `ByteMessage`.

Example

Example 4.6 on page 77 shows the WSDL for configuring a JMS consumer endpoint.

Example 4.6. WSDL for a JMS consumer endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:client messageType="binary" />
  </port>
</service>
```

JMS provider configuration

Overview

JMS provider endpoints have a number of behaviors that are configurable. These include:

- how messages are correlated
- the use of durable subscriptions
- if the service uses local JMS transactions
- the message selectors used by the endpoint

Specifying the configuration

Provider endpoint behaviors are configured using the optional `jms:server` element. The `jms:server` element is a child of the WSDL `wsdl:port` element and has the following attributes:

Table 4.4. JMS provider endpoint WSDL extensions

Attribute	Description
`useMessageIDAsCorrealationID`	Specifies whether JMS will use the message ID to correlate messages. The default is `false`.
`durableSubscriberName`	Specifies the name used to register a durable subscription.
`messageSelector`	Specifies the string value of a message selector to use. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false`. [a]

[a] Currently, setting the `transactional` attribute to `true` is not supported by the runtime.

Example

Example 4.7 on page 79 shows the WSDL for configuring a JMS provider endpoint.

Example 4.7. WSDL for a JMS provider endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:server messageSelector="cxf_message_selector"
                useMessageIDAsCorrelationID="true"
                transactional="true"
                durableSubscriberName="cxf_subscriber" />
  </port>
</service>
```

Using a Named Reply Destination

Overview

By default, Fuse Services Framework endpoints using JMS create a temporary queue for sending replies back and forth. If you prefer to use named queues, you can configure the queue used to send replies as part of an endpoint's JMS configuration.

Setting the reply destination name

You specify the reply destination using either the `jmsReplyDestinationName` attribute or the `jndiReplyDestinationName` attribute in the endpoint's JMS configuration. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the `ReplyTo` field of all outgoing requests. A service endpoint will use the value of the `jndiReplyDestinationName` attribute as the location for placing replies if there is no destination specified in the request's `ReplyTo` field.

Example

Example 4.8 on page 80 shows the configuration for a JMS client endpoint.

Example 4.8. JMS Consumer Specification Using a Named Reply Queue

```
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
                 jndiConnectionFactoryName="myConnectionFactory"
                 jndiDestinationName="myDestination"
                 jndiReplyDestinationName="myReplyDestination" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.apache.cxf.transport.jms.MyInitialContextFactory"
/>
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </jms:conduit>
```

Chapter 5. Fuse Services Framework Logging

This chapter describes how to configure logging in the Fuse Services Framework runtime.

Overview of Fuse Services Framework Logging

Overview

Fuse Services Framework 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

Fuse Services Framework 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

Fuse Services Framework includes a logging feature that can be plugged into your client or your service to enable logging. Example 5.1 on page 82 shows the configuration to enable the logging feature.

Example 5.1. 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 93.

Where to begin?

To run a simple example of logging follow the instructions outlined in a "Simple Example of Using Logging" on page 84.

For more information on how logging works in Fuse Services Framework, 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 Fuse Services Framework 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`**

Platform	Command
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 configuration file

The default logging configuration file, `logging.properties`, is located in the *`InstallDir`*`/etc` directory. It configures the Fuse Services Framework 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 5.1 on page 87 shows the handlers that are configured in the default `logging.properties` file.

Table 5.1. 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 Fuse Services Framework environment.

Configuring the console handler

Example 5.2 on page 87 shows the code for configuring the console logger.

Example 5.2. Configuring the Console Handler

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

The console handler also supports the configuration properties shown in Example 5.3 on page 87.

Example 5.3. 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 5.3 on page 87 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 89). 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 5.4 on page 88 shows code that configures the file handler.

Example 5.4. Configuring the File Handler

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

The file handler also supports the configuration properties shown in Example 5.5 on page 88.

Example 5.5. 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 5.5 on page 88 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 5.6 on page 88.

Example 5.6. 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 5.7 on page 89.

Example 5.7. 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 5.8 on page 89 configures logging at a `SEVERE` level on classes in the `com.xyz.foo` package.

Example 5.8. 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 5.9 on page 90 when starting the application.

Example 5.9. 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 89 to set fine-grained logging for specified Fuse Services Framework logging subsystems.

Fuse Services Framework logging subsystems

Table 5.2 on page 91 shows a list of available Fuse Services Framework logging subsystems.

Table 5.2. Fuse Services Framework Logging Subsystems

Subsystem	Description
`org.apache.cxf.aegis`	Aegis binding
`org.apache.cxf.binding.coloc`	colocated binding
`org.apache.cxf.binding.http`	HTTP binding
`org.apache.cxf.binding.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`	Fuse Services Framework 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
`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

Subsystem	Description
`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 5.10 on page 92.

Example 5.10. 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 5.10 on page 92 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 5.11 on page 93.

Example 5.11. 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 5.11 on page 93 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 5.12 on page 93.

Example 5.12. 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 5.12 on page 93 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 5.13 on page 94.

Example 5.13. 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 5.14 on page 94 to the `cxf.xml` configuration file located in the wsdl_first sample's directory:

Example 5.14. 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.

Chapter 6. Deploying WS-Addressing

Fuse Services Framework supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Fuse Services Framework 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

Fuse Services Framework 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 Fuse Services Framework, WS-Addressing functionality is implemented as interceptors. The Fuse Services Framework 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 6.1 on page 99.

Table 6.1. 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:

- Fuse Services Framework 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 6.1 on page 100 and Example 6.2 on page 100 respectively.

Example 6.1. client.xml—Adding WS-Addressing Feature to Client Configuration

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

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

Example 6.2. 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.springframe
work.org/schema/beans/spring-beans.xsd">
```

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

Configuring WS-Addressing Attributes

Overview

The Fuse Services Framework WS-Addressing feature element is defined in the namespace http://cxf.apache.org/ws/addressing. It supports the two attributes described in Table 6.2 on page 102.

Table 6.2. 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 6.3 on page 102 an addressing policy assertion to enable non-anonymous responses is embedded in the `policies` element.

Example 6.3. 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>
```

Chapter 7. Enabling Reliable Messaging

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

Introduction to WS-RM

Overview

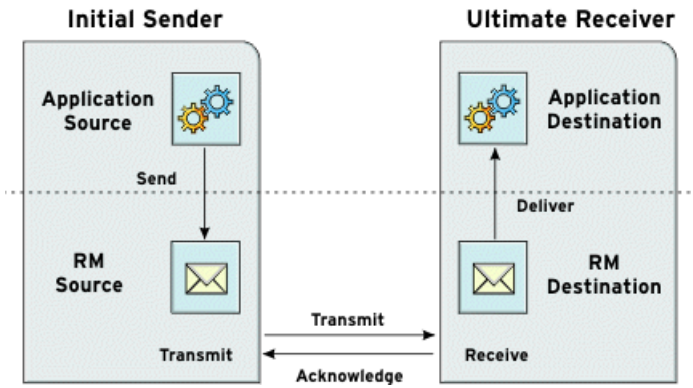
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

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 7.1 on page 106.

Figure 7.1. 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 114.

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

Fuse Services Framework 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 Fuse Services Framework, WS-RM functionality is implemented as interceptors. The Fuse Services Framework 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.

Fuse Services Framework WS-RM Interceptors

The Fuse Services Framework WS-RM implementation consists of four interceptors, which are described in Table 7.1 on page 108.

Table 7.1. Fuse Services Framework WS-ReliableMessaging Interceptors

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

Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, 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 110.

Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Fuse Services Framework 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 114.

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 Fuse Services Framework 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 Fuse Services Framework 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 7.1 on page 110).

Example 7.1. 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 7.1 on page 110 can be explained as follows:

❶ A Fuse Services Framework 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 97.

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 Fuse Services Framework WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. Example 7.2 on page 112 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 7.2. 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 7.3 on page 112.

Example 7.3. Adding an RM Policy to Your WSDL File

```
<wsp:Policy wsu:Id="RM"
   xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd">
    <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
    </wsam:Addressing>
    <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
```

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

```
    </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 Fuse Services Framework-specific attributes that are defined in the Fuse Services Framework 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 Fuse Services Framework-Specific WS-RM Attributes

Overview

To configure the Fuse Services Framework-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 7.4 on page 115 shows a simple example.

Example 7.4. Configuring Fuse Services Framework-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.springframework.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd">
...
<wsrm-mgr:rmManager>
<!--
  ...Your configuration goes here
-->
</wsrm-mgr:rmManager>
```

Children of the rmManager Spring bean

Table 7.2 on page 115 shows the child elements of the `rmManager` Spring bean, defined in the http://cxf.apache.org/ws/rm/manager namespace.

Table 7.2. 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 123.

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 7.3 on page 117 shows the elements defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace:

Table 7.3. Children of the WS-Policy RMAssertion Element

Name	Description
`InactivityTimeout`	Specifies the amount of time that must pass without receiving a message before an endpoint can consider an RM sequence to have been terminated due to inactivity.
`BaseRetransmissionInterval`	Sets the interval within which an acknowledgement must be received by the RM Source for a given message. If an acknowledgement is not received within the time set by the `BaseRetransmissionInterval`, the RM Source will retransmit the message.
`ExponentialBackoff`	Indicates the retransmission interval will be adjusted using the commonly known exponential backoff algorithm (Tanenbaum). For more information, see *Computer Networks*, Andrew S. Tanenbaum, Prentice Hall PTR, 2003.
`AcknowledgementInterval`	In WS-RM, acknowledgements are sent on return messages or sent stand-alone. If a return message is not available to send an acknowledgement, an RM Destination can wait for up to the acknowledgement interval before sending a stand-alone

Name	Description
	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 Fuse Services Framework `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 Fuse Services Framework-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 7.5 on page 118 shows:

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

Example 7.5. 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 7.6 on page 119.

Example 7.6. 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 a 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 112 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 7.7 on page 120 shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 7.7. 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 117.

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 7.8 on page 121 shows how to set the WS-RM base retransmission interval.

Example 7.8. 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 7.9 on page 122 shows how to set the WS-RM exponential backoff for retransmission.

Example 7.9. 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 7.10 on page 122 shows how to set the WS-RM acknowledgement interval.

Example 7.10. 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 7.11 on page 123 shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 7.11. 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 newsequence.

Example 7.12 on page 123 shows how to set the maximum length of an RM sequence.

Example 7.12. 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 7.13 on page 124 shows how to set the WS-RM message delivery assurance.

Example 7.13. 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 Fuse Services Framework 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 an 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.

Fuse Services Framework enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Fuse Services Framework includes Derby for out-of-the-box deployment. In addition, the persistent store is also exposed using a Java API.

Important

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

How it works

Fuse Services Framework 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 Fuse Services Framework.

The configuration shown in Example 7.14 on page 126 enables the JDBC-based store that comes with Fuse Services Framework.

Example 7.14. Configuration for the Default WS-RM Persistence Store

```
<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 Fuse Services Framework supports the properties shown in Table 7.4 on page 126.

Table 7.4. 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 7.15 on page 126 enables the JDBC-based store that comes with Fuse Services Framework, while setting the `driverClassName` and `url` to non-default values.

Example 7.15. 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>
```

Chapter 8. Enabling High Availability

This chapter explains how to enable and configure high availability in the Fuse Services Framework 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 Fuse Services Framework delivers the requests to one of the member replicas. The routing to a replica is transparent to the client.

HA with static failover

Fuse Services Framework supports high availability (HA) with static failover in which replica details are encoded in the service WSDL file. The WSDL file contains multiple ports, and can contain 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.

Enabling HA with Static Failover

Overview

To enable HA with static failover, you must do the following:

1. "Encode replica details in your service WSDL file"
2. "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 8.1 on page 129 shows a WSDL file extract that defines a service cluster of three replicas.

Example 8.1. 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 8.1 on page 129 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 8.2 on page 130.

Example 8.2. 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 Fuse Services Framework'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 random replica service 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, add the configuration shown in Example 8.3 on page 131 to your client configuration file.

Example 8.3. 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 8.3 on page 131 can be explained as follows:

❶ Defines a `Random` bean and implementation class that implements the random strategy.
❷ Specifies that the random strategy is used when selecting a replica.

Chapter 9. Packaging an Application

Applications must be packed as an OSGi bundle before they can be deployed into Fuse ESB. You will not need to include any Fuse Service Framework specific packages in your bundle. The Fuse Service Framework packages are included in Fuse ESB. You need to ensure you import the required packages when building your bundle.

Creating a bundle

To deploy a Fuse Service Framework application into Fuse ESB, you need to package it as an OSGi bundle. There are several tools available for assisting in the process. Fuse ESB uses the Maven bundle plug-in whose use is described in Appendix B on page 139.

Required bundle

The Fuse Service Framework runtime components are included in Fuse ESB as an OSGi bundle called `org.apache.cxf.cxf-bundle`. This bundle needs to be installed in the Fuse ESB container before your application's bundle can be started.

To inform the container of this dependency, you use the OSGi manifest's `Required-Bundle` property.

Required packages

In order for your application to use the Fuse Service Framework components, you need to import their packages into the application's bundle. Because of the complex nature of the dependencies in Fuse Service Framework, you cannot rely on the Maven bundle plug-in, or 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.apache.servicemix.cxf.transport.http_osgi`
- `org.springframework.beans.factory.config`

Example

Example 9.1 on page 134 shows a manifest for a Fuse Service Framework application's OSGi bundle.

Example 9.1. Fuse Service Framework Application Manifest

```
Manifest-Version: 1.0
Built-By: FinnMcCumial
Created-By: Apache Maven Bundle Plugin
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0.txt
Import-Package: javax.jws,javax.wsdl,META-INF.cxf,META-INF.cxf.osgi,
org.apache.cxf.bus,org.apache.cxf.bus.spring,org.apache.bus.resource,
org.apache.cxf.configuration.spring, org.apache.cxf.resource,
org.apache.servicemix.cxf.transport.http_cxf,
org.springframework.beans.factory.config
Bnd-LastModified: 1222079507224
Bundle-Version: 4.0.0.fuse
Bundle-Name: Fuse CXF Example
Bundle-Description: This is a sample CXF manifest.
Build-Jdk: 1.5.0_08
Private-Package: org.apache.servicemix.examples.cxf
Required-Bundle: org.apache.cxf.cxf-bundle
Bundle-ManifestVersion: 2
Bundle-SymbolicName: cxf-wsdl-first-osgi
Tool: Bnd-0.0.255
```

Chapter 10. Deploying an Application

Fuse ESB will automatically install and deploy your application. You can also manually control the state of your application using the console.

Overview

There are two ways to deploy your application into Fuse ESB:

1. Rely on the hot deployment mechanism.
2. Use the console.

You can also start and stop a deployed application using the console.

Hot deployment

The easiest way to deploy an application is to place it in the hot deployment folder. By default, the hot deployment folder is *`InstallDir`*`/deploy`. Any bundle placed in this folder is installed into the container. If its dependencies can be resolved, the bundle is activated.

One the bundle is installed in the container, you can manage it using the console.

Deploying from the console

The easiest way to deploy an application from the console is to install it and start it in one step. This is done using the **osgi install -s** command. It takes the location of the bundle as a URI. So the command:

```
servicemix>osgi install -s file:/home/finn/ws/widgetapp.jar
```

Installs and attempts to start the bundle `widgetapp.jar` which is located in `/home/finn/ws`.

You can use the **osgi install** command without the `-s` flag. That will install the bundle without attempting to start it. You will then have to manually start the bundle using the **osgi start** command.

The **osgi start** command uses the bundle ID to determine which bundle to activate. [1]

Refreshing an application

If you make changes to your application and want to redeploy it, you can do so by replacing the installed bundle with a new version and using the **osgi refresh** command. This command instructs the container to stop the running instance of your application, reload the bundle, and restart it.

[1] You can get a list of the bundle IDs using the **osgi list** command.

The **osgi refresh** command uses a bundle ID to determine which bundle to refresh. [1]

Stopping an application

If you want to temporarily deactivate your application you can use the **osgi stop** command. The **osgi stop** moves your application's bundle from the active state to the resolved state. This means that it can be easily restarted using the **osgi start** command.

The **osgi stop** command uses a bundle ID to determine which bundle to stop. [1]

Uninstalling an application

When you want to permanently remove an application from the container you need to uninstall it. Bundles can only be installed when they are not active. This means that you have to stop your application using the **osgi stop** command before trying to unistall it.

Once the application's bundle is stopped, you can use the **osgi uninstall** command to remove the bundle from the container. This does not delete the physical bundle. It just removes the bundle from the container's list of installed bundles.

The **osgi stop** command uses a bundle ID to determine which bundle to unistall. [1]

Appendix A. Fuse Services Framework 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

Appendix B. Using the Maven OSGi Tooling

Manually creating a bundle, or a collection of bundles, for a large project can be cumbersome. The Maven bundle plug-in makes the job easier by automating the process and providing a number of shortcuts for specifying the contents of the bundle manifest.

The Fuse ESB OSGi tooling uses the Maven bundle plug-in[1] from Apache Felix. The bundle plug-in is based on the **bnd**[2] 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 plug-in can calculate the proper values to populate the `Import-Packages` and the `Export-Package` properties in the bundle manifest. The plug-in also has default values that are used for other required properties in the bundle manifest.

To use the bundle plug-in, do the following:

1. Add the bundle plug-in to your project's POM file.
2. Configure the plug-in to correctly populate your bundle's manifest.

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

Setting up a Fuse ESB OSGi project

Overview

A Maven project for building an OSGi bundle can be a simple single level project. It does not require any sub-projects. However, it does require that you do the following:

1. Add the bundle plug-in to your POM.
2. Instruct Maven to package the results as an OSGi bundle.

Tip

There are several Maven archetypes you can use to set up your project with the appropriate settings.

Directory structure

A project that constructs an OSGi bundle can be a single level project. It only requires that you have a top-level POM file and a `src` folder. As in all Maven projects, you place all Java source code in the `src/java` folder, and you place any non-Java resources in the `src/resources` folder.

Non-Java resources include Spring configuration files, JBI endpoint configuration files, and WSDL contracts.

Note

Fuse ESB OSGi projects that use Fuse Service Framework, Fuse Mediation Router, or another Spring configured bean also include a `beans.xml` file located in the `src/resources/META-INF/spring` folder.

Adding a bundle plug-in

Before you can use the bundle plug-in you must add a dependency on Apache Felix. After you add the dependency, you can add the bundle plug-in to the plug-in portion of the POM.

Example B.1 on page 141 shows the POM entries required to add the bundle plug-in to your project.

Example B.1. Adding an OSGi bundle plug-in to a POM

```
...
<dependencies>
  <dependency> ❶
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin> ❷
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName> ❸
          <Import-Package>*,org.apache.camel.osgi</Import-Package> ❹
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package> ❺
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example B.1 on page 141 do the following:

❶ Adds the dependency on Apache Felix
❷ Adds the bundle plug-in to your project
❸ Configures the plug-in to use the project's artifact ID as the bundle's symbolic name
❹ Configures the plug-in to include all Java packages imported by the bundled classes; also imports the `org.apache.camel.osgi` package
❺ Configures the plug-in to bundle the listed class, but not to include them in the list of exported packages

Note

Edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see "Configuring the Bundle Plug-In" on page 145.

Activating a bundle plug-in

To have Maven use the bundle plug-in, instruct it to package the results of the project as a bundle. Do this by setting the POM file's `packaging` element to `bundle`.

Useful Maven archetypes

There are several Maven archetypes to generate a project that is preconfigured to use the bundle plug-in:

- "Spring OSGi archetype"
- "Fuse Service Framework code-first archetype"
- "Fuse Service Framework wsdl-first archetype"
- "Fuse Mediation Router archetype"

Spring OSGi archetype

The Spring OSGi archetype creates a generic project for building an OSGi project using Spring DM, as shown:

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.springframework.osgi
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=1.12
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Fuse Service Framework code-first archetype

The Fuse Service Framework code-first archetype creates a project for building a service from Java, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Fuse Service Framework wsdl-first archetype

The Fuse Service Framework wsdl-first archetype creates a project for creating a service from WSDL, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Fuse Mediation Router archetype

The Fuse Mediation Router archetype creates a project for building a route that is deployed into Fuse ESB, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-camel-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Configuring the Bundle Plug-In

Overview

A bundle plug-in requires very little information to function. All of the required properties use default settings to generate a valid OSGi bundle.

While you can create a valid bundle using just the default values, you will probably want to modify some of the values. You can specify most of the properties inside the plug-in's `instructions` element.

Configuration properties

Some of the commonly used configuration properties are:

- Bundle-SymbolicName
- Bundle-Name
- Bundle-Version
- Export-Package
- Private-Package
- Import-Package

Setting a bundle's symbolic name

By default, the bundle plug-in sets the value for the `Bundle-SymbolicName` property to *`groupId`* `+ "." +` *`artifactId`*, with the following exceptions:

- If *`groupId`* has only one section (no dots), the first package name with classes is returned.

  For example, if the group Id is `commons-logging:commons-logging`, the bundle's symbolic name is `org.apache.commons.logging`.

- If *`artifactId`* is equal to the last section of *`groupId`*, then *`groupId`* is used.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven`, the bundle's symbolic name is `org.apache.maven`.

- If *`artifactId`* starts with the last section of *`groupId`*, that portion is removed.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven-core`, the bundle's symbolic name is `org.apache.maven.core`.

To specify your own value for the bundle's symbolic name, add a `Bundle-SymbolicName` child in the plug-in's `instructions` element, as shown in Example B.2.

Example B.2. Setting a bundle's symbolic name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to `${project.name}`.

To specify your own value for the bundle's name, add a `Bundle-Name` child to the plug-in's `instructions` element, as shown in Example B.3.

Example B.3. Setting a bundle's name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's version

By default, a bundle's version is set to `${project.version}`. Any dashes (`-`) are replaced with dots (`.`) and the number is padded up to four digits. For example, `4.2-SNAPSHOT` becomes `4.2.0.SNAPSHOT`.

To specify your own value for the bundle's version, add a `Bundle-Version` child to the plug-in's `instructions` element, as shown in Example B.4.

Example B.4. Setting a bundle's version

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying exported packages

By default, the OSGi manifest's `Export-Package` list is populated by all of the packages in your local Java source code (under `src/main/java`), *except* for the deault package, `.`, and any packages containing `.impl` or `.internal`.

Important

If you use a `Private-Package` element in your plug-in configuration and you do not specify a list of packages to export, the default behavior includes only the packages listed in the `Private-Package` element in the bundle. No packages are exported.

The default behavior can result in very large packages and in exporting packages that should be kept private. To change the list of exported packages you can add an `Export-Package` child to the plug-in's `instructions` element.

The `Export-Package` element specifies a list of packages that are to be included in the bundle and that are to be exported. The package names can be specified using the `*` wildcard symbol. For example, the entry `com.fuse.demo.*` includes all packages on the project's classpath that start with `com.fuse.demo`.

You can specify packages to be excluded be prefixing the entry with `!`. For example, the entry `!com.fuse.demo.private` excludes the package `com.fuse.demo.private`.

When excluding packages, the order of entries in the list is important. The list is processed in order from the beginning and any subsequent contradicting entries are ignored.

For example, to include all packages starting with `com.fuse.demo` except the package `com.fuse.demo.private`, list the packages using:

```
!com.fuse.demo.private,com.fuse.demo.*
```

However, if you list the packages using `com.fuse.demo.*,!com.fuse.demo.private`, then `com.fuse.demo.private` is included in the bundle because it matches the first pattern.

Specifying private packages

If you want to specify a list of packages to include in a bundle *without* exporting them, you can add a `Private-Package` instruction to the bundle plug-in configuration. By default, if you do not specify a `Private-Package` instruction, all packages in your local Java source are included in the bundle.

Important

If a package matches an entry in both the `Private-Package` element and the `Export-Package` element, the `Export-Package` element takes precedence. The package is added to the bundle and exported.

The `Private-Package` element works similarly to the `Export-Package` element in that you specify a list of packages to be included in the bundle. The bundle plug-in uses the list to find all classes on the project's classpath that are to be included in the bundle. These packages are packaged in the bundle, but not exported (unless they are also selected by the `Export-Package` instruction).

Example B.5 shows the configuration for including a private package in a bundle

Example B.5. Including a private package in a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying imported packages

By default, the bundle plug-in populates the OSGi manifest's `Import-Package` property with a list of all the packages referred to by the contents of the bundle.

While the default behavior is typically sufficient for most projects, you might find instances where you want to import packages that are not automatically added to the list. The default behavior can also result in unwanted packages being imported.

To specify a list of packages to be imported by the bundle, add an `Import-Package` child to the plug-in's `instructions` element. The syntax for the package list is the same as for the `Export-Package` element and the `Private-Package` element.

Important

When you use the `Import-Package` element, the plug-in does not automatically scan the bundle's contents to determine if there are any required imports. To ensure that the contents of the bundle are scanned, you must place an `*` as the last entry in the package list.

Example B.6 shows the configuration for specifying the packages imported by a bundle

Example B.6. Specifying the packages imported by a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws,
         javax.wsdl,
         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,
         *
     </Import-Package>
     ...
   </instructions>
  </configuration>
</plugin>
```

More information

For more information on configuring a bundle plug-in, see:

- Apache Felix documentation[3]
- Peter Kriens' aQute Software Consultancy web site[4]

[3] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html
[4] http://www.aqute.biz/Code/Bnd

Appendix C. Conduits

Conduits are a low-level piece of the transport architecture that are used to implement outbound connections. Their behavior and life-cycle can effect system performance and processing load.

Overview

Conduits manage the client-side, or outbound, transport details in the Fuse Services Framework runtime. They are responsible for opening ports, establishing outbound connections, sending messages, and listening for any responses between an application and a single external endpoint. If an application connects to multiple endpoints, it will have one conduit instance for each endpoint.

Each transport type implements its own conduit using the `Conduit` interface. This allows for a standardized interface between the application level functionality and the transports.

In general, you only need to worry about the conduits being used by your application when configuring the client-side transport details. The underlying semantics of how the runtime handles conduits is, generally, not something a developer needs to worry about.

However, there are cases when an understanding of conduit's can prove helpful:

- Implementing a custom transport
- Advanced application tuning to manage limited resources

Conduit life-cycle

Conduits are managed by the client implementation object. Once created, a conduit lives for the duration of the client implementation object. The conduit's life-cycle is:

1. When the client implementation object is created, it is given a reference to a `ConduitSelector` object.
2. When the client needs to send a message is request's a reference to a conduit from the conduit selector.

   If the message is for a new endpoint, the conduit selector creates a new conduit and passes it to the client implementation. Otherwise, it passes the client a reference to the conduit for the target endpoint.
3. The conduit sends messages when needed.
4. When the client implementation object is destroyed, all of the conduits associated with it are destroyed.

Conduit weight

The weight of a conduit object depends on the transport implementation. HTTP conduits are extremely light weight. JMS conduits are heavy because they are associated with the JMS `Session` object and one or more `JMSListenerContainer` objects.

Index

I

J

M

N

O

P

R

S

U

W