Fuse ESB Enterprise

Using the Web Services Bindings and Transports

Version 7.1
December 2012

Integration Everywhere

Using the Web Services Bindings and Transports

Version 7.1

Updated: 08 Jan 2014

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Fuse ESB Enterprise 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

Part I. Bindings

Chapter 1. Understanding Bindings in WSDL

Bindings map the logical messages used to define a service into a concrete payload format that can be transmitted and received by an endpoint.

Overview

Bindings provide a bridge between the logical messages used by a service to a concrete data format that an endpoint uses in the physical world. They describe how the logical messages are mapped into a payload format that is used on the wire by an endpoint. It is within the bindings that details such as parameter order, concrete data types, and return values are specified. For example, the parts of a message can be reordered in a binding to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

Port types and bindings

Port types and bindings are directly related. A port type is an abstract definition of a set of interactions between two logical services. A binding is a concrete definition of how the messages used to implement the logical services will be instantiated in the physical world. Each binding is then associated with a set of network details that finish the definition of one endpoint that exposes the logical service defined by the port type.

To ensure that an endpoint defines only a single service, WSDL requires that a binding can only represent a single port type. For example, if you had a contract with two port types, you could not write a single binding that mapped both of them into a concrete data format. You would need two bindings.

However, WSDL allows for a port type to be mapped to several bindings. For example, if your contract had a single port type, you could map it into two or more bindings. Each binding could alter how the parts of the message are mapped or they could specify entirely different payload formats for the message.

The WSDL elements

Bindings are defined in a contract using the WSDL `binding` element. The binding element has a single attribute, `name`, that specifies a unique name for the binding. The value of this attribute is used to associate the binding with an endpoint as discussed in "Defining Your Logical Interfaces" in *Writing WSDL Contracts*.

The actual mappings are defined in the children of the `binding` element. These elements vary depending on the type of payload format you decide to use. The different payload formats and the elements used to specify their mappings are discussed in the following chapters.

Adding to a contract

Fuse Services Framework provides command line tools that can generate bindings for predefined service interfaces.

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different types of bindings work.

You can also add a binding to a contract using any text editor. When hand editing a contract, you are responsible for ensuring that the contract is valid.

Supported bindings

The Fuse Services Framework supports the following bindings:

- SOAP 1.1
- SOAP 1.2
- CORBA
- Pure XML

Chapter 2. Using SOAP 1.1 Messages

Fuse Services Framework provides a tool to generate a SOAP 1.1 binding which does not use any SOAP headers. However, you can add SOAP headers to your binding using any text or XML editor.

Adding a SOAP 1.1 Binding

Note

To use wsdl2soap you will need to download the Fuse Services Framework distribution.

Using wsdl2soap

To generate a SOAP 1.1 binding using **wsdl2soap** use the following command:

`wsdl2soap` {-i *`port-type-name`*} [-b *`binding-name`*] [-d *`output-directory`*] [-o *`output-file`*] [-n *`soap-body-namespace`*] [-style (document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] *`wsdlurl`*

The command has the following options:

Option	Interpretation
`-i` *`port-type-name`*	Specifies the `portType` element for which a binding is generated.
`wsdlurl`	The path and name of the WSDL file containing the `portType` element definition.

The tool has the following optional arguments:

Option	Interpretation
`-b` *`binding-name`*	Specifies the name of the generated SOAP binding.
`-d` *`output-directory`*	Specifies the directory to place the generated WSDL file.
`-o` *`output-file`*	Specifies the name of the generated WSDL file.
`-n` *`soap-body-namespace`*	Specifies the SOAP body namespace when the style is RPC.
`-style` (`document`/`rpc`)	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is `document`.

Option	Interpretation
`-use (literal/encoded)`	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is `literal`.
`-v`	Displays the version number for the tool.
`-verbose`	Displays comments during the code generation process.
`-quiet`	Suppresses comments during the code generation process.

The `-i` *`port-type-name`* and *`wsdlurl`* arguments are required. If the `-style rpc` argument is specified, the `-n` *`soap-body-namspace`* argument is also required. All other arguments are optional and may be listed in any order.

Important

wsdl2soap does not support the generation of `document/encoded` SOAP bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a WSDL fragment similar to the one shown in Example 2.1 on page 19.

Example 2.1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
```

```
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

The SOAP binding generated for `orderWidgets` is shown in Example 2.2 on page 20.

Example 2.2. SOAP 1.1 Binding for `orderWidgets`

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the `document/literal` message style.

Adding SOAP Headers to a SOAP 1.1 Binding

Overview

SOAP headers are defined by adding `soap:header` elements to your default SOAP 1.1 binding. The `soap:header` element is an optional child of the `input`, `output`, and `fault` elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many SOAP headers as needed.

Syntax

The syntax for defining a SOAP header is shown in Example 2.3 on page 21. The `message` attribute of `soap:header` is the qualified name of the message from which the part being inserted into the header is taken. The `part` attribute is the name of the message part inserted into the SOAP header. Because SOAP headers are always document style, the WSDL message part inserted into the SOAP header must be defined using an element. Together the `message` and the `part` attributes fully describe the data to insert into the SOAP header.

Example 2.3. SOAP Header Syntax

```
<binding name="headwig">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap:operation soapAction="" style="document"/>
      <input name="grain">
        <soap:body ... />
        <soap:header message="QName" part="partName"/>
      </input>
...
</binding>
```

As well as the mandatory `message` and `part` attributes, `soap:header` also supports the `namespace`, the `use`, and the `encodingStyle` attributes. These optional attributes function the same for `soap:header` as they do for `soap:body`.

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would want to send information twice in the same message, the SOAP binding provides

a means for specifying the message parts that are inserted into the SOAP body.

The `soap:body` element has an optional attribute, `parts`, that takes a space delimited list of part names. When `parts` is defined, only the message parts listed are inserted into the SOAP body. You can then insert the remaining parts into the SOAP header.

Note

When you define a SOAP header using parts of the parent message, Fuse Services Framework automatically fills in the SOAP headers for you.

Example

Example 2.4 on page 22 shows a modified version of the `orderWidgets` service shown in Example 2.1 on page 19. This version has been modified so that each order has an xsd:base64binary value placed in the SOAP header of the request and response. The SOAP header is defined as being the `keyVal` part from the `widgetKey` message. In this case you are responsible for adding the SOAP header to your application logic because it is not part of the input or output message.

Example 2.4. SOAP 1.1 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
```

```
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

You can modify Example 2.4 on page 22 so that the header value is a part of the input and output messages as shown in Example 2.5 on page 23. In this case `keyVal` is a part of the input and output messages. In the `soap:body` element's `parts` attribute specifies that `keyVal` cannot be inserted into the body. However, it is inserted into the SOAP header.

Example 2.5. SOAP 1.1 Binding for orderWidgets with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
```

```
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal" parts="numOrdered"/>
        <soap:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap:body use="literal" parts="bill"/>
        <soap:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
```

```
      </fault>
  </operation>
</binding>
...
</definitions>
```

Chapter 3. Using SOAP 1.2 Messages

Fuse Services Framework provides tools to generate a SOAP 1.2 binding which does not use any SOAP headers. You can add SOAP headers to your binding using any text or XML editor.

Adding a SOAP 1.2 Binding to a WSDL Document

Using wsdl2soap

Note

To use wsdl2soap you will need to download the Fuse Services Framework distribution.

To generate a SOAP 1.2 binding using **wsdl2soap** use the following command:

`wsdl2soap` {-i *`port-type-name`*} [-b *`binding-name`*] {-soap12} [-d *`output-directory`*] [-o *`output-file`*] [-n *`soap-body-namespace`*] [-style (document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] *`wsdlurl`*

The tool has the following required arguments:

Option	Interpretation
`-i` *`port-type-name`*	Specifies the `portType` element for which a binding is generated.
`-soap12`	Specifies that the generated binding uses SOAP 1.2.
`wsdlurl`	The path and name of the WSDL file containing the `portType` element definition.

The tool has the following optional arguments:

Option	Interpretation
`-b` *`binding-name`*	Specifies the name of the generated SOAP binding.
`-soap12`	Specifies that the generated binding will use SOAP 1.2.
`-d` *`output-directory`*	Specifies the directory to place the generated WSDL file.
`-o` *`output-file`*	Specifies the name of the generated WSDL file.
`-n` *`soap-body-namespace`*	Specifies the SOAP body namespace when the style is RPC.

Option	Interpretation
`-style` (`document`/`rpc`)	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is `document`.
`-use` (`literal`/`encoded`)	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is `literal`.
`-v`	Displays the version number for the tool.
`-verbose`	Displays comments during the code generation process.
`-quiet`	Suppresses comments during the code generation process.

The `-i` *`port-type-name`* and *`wsdlurl`* arguments are required. If the `-style rpc` argument is specified, the `-n` *`soap-body-namspace`* argument is also required. All other arguments are optional and can be listed in any order.

Important

wsdl2soap does not support the generation of `document/encoded` SOAP 1.2 bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a WSDL fragment similar to the one shown in Example 3.1 on page 29.

Example 3.1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
```

```
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

The SOAP binding generated for `orderWidgets` is shown in Example 3.2 on page 30.

Example 3.2. SOAP 1.2 Binding for orderWidgets

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
      </input>
      <output name="bill">
        <wsoap12:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the `document/literal` message style.

Adding Headers to a SOAP 1.2 Message

Overview

SOAP message headers are defined by adding `soap12:header` elements to your SOAP 1.2 message. The `soap12:header` element is an optional child of the `input`, `output`, and `fault` elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many headers as needed.

Syntax

The syntax for defining a SOAP header is shown in Example 3.3 on page 31.

Example 3.3. SOAP Header Syntax

```
<binding name="headwig">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap12:operation soapAction="" style="documment"/>
      <input name="grain">
        <soap12:body ... />
        <soap12:header message="QName" part="partName"
                       use="literal|encoded"
                        encodingStyle="encodingURI"
                        namespace="namespaceURI" />
      </input>
...
</binding>
```

The `soap12:header` element's attributes are described in Table 3.1 on page 31.

Table 3.1. `soap12:header` Attributes

Attribute	Description
`message`	A required attribute specifying the qualified name of the message from which the part being inserted into the header is taken.
`part`	A required attribute specifying the name of the message part inserted into the SOAP header.
`use`	Specifies if the message parts are to be encoded using encoding rules. If set to `encoded` the message parts are encoded using the encoding rules specified by the value of the `encodingStyle` attribute. If set to `literal`, the

Attribute	Description
	message parts are defined by the schema types referenced.
`encodingStyle`	Specifies the encoding rules used to construct the message.
`namespace`	Defines the namespace to be assigned to the header element serialized with `use="encoded"`.

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would send information twice in the same message, the SOAP 1.2 binding provides a means for specifying the message parts that are inserted into the SOAP body.

The `soap12:body` element has an optional attribute, `parts`, that takes a space delimited list of part names. When `parts` is defined, only the message parts listed are inserted into the body of the SOAP 1.2 message. You can then insert the remaining parts into the message's header.

Note

When you define a SOAP header using parts of the parent message, Fuse Services Framework automatically fills in the SOAP headers for you.

Example

Example 3.4 on page 32 shows a modified version of the `orderWidgets` service shown in Example 3.1 on page 29. This version is modified so that each order has an xsd:base64binary value placed in the header of the request and the response. The header is defined as being the `keyVal` part from the `widgetKey` message. In this case you are responsible for adding the application logic to create the header because it is not part of the input or output message.

Example 3.4. SOAP 1.2 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
```

```
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
```

```
  </operation>
</binding>
...
</definitions>
```

You can modify Example 3.4 on page 32 so that the header value is a part of the input and output messages, as shown in Example 3.5 on page 34. In this case `keyVal` is a part of the input and output messages. In the `soap12:body` elements the `parts` attribute specifies that `keyVal` should not be inserted into the body. However, it is inserted into the header.

Example 3.5. SOAP 1.2 Binding for orderWidgets with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
```

```
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal" parts="numOrdered"/>
        <soap12:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal" parts="bill"/>
        <soap12:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

Chapter 4. Sending Binary Data Using SOAP with Attachments

SOAP attachments provide a mechanism for sending binary data as part of a SOAP message. Using SOAP with attachments requires that you define your SOAP messages as MIME multipart messages.

Overview

SOAP messages generally do not carry binary data. However, the W3C SOAP 1.1 specification allows for using MIME multipart/related messages to send binary data in SOAP messages. This technique is called using SOAP with attachments. SOAP attachments are defined in the W3C's *SOAP Messages with Attachments Note*[1].

Namespace

The WSDL extensions used to define the MIME multipart/related messages are defined in the namespace `http://schemas.xmlsoap.org/wsdl/mime/`.

In the discussion that follows, it is assumed that this namespace is prefixed with mime. The entry in the WSDL `definitions` element to set this up is shown in Example 4.1 on page 37.

Example 4.1. MIME Namespace Specification in a Contract

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

Changing the message binding

In a default SOAP binding, the first child element of the `input`, `output`, and `fault` elements is a `soap:body` element describing the body of the SOAP message representing the data. When using SOAP with attachments, the `soap:body` element is replaced with a `mime:multipartRelated` element.

Note

WSDL does not support using `mime:multipartRelated` for `fault` messages.

The `mime:multipartRelated` element tells Fuse Services Framework that the message body is a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their

[1] http://www.w3.org/TR/SOAP-attachments

contents. `mime:multipartRelated` elements contain one or more `mime:part` elements that describe the individual parts of the message.

The first `mime:part` element must contain the `soap:body` element that would normally appear in a default SOAP binding. The remaining `mime:part` elements define the attachments that are being sent in the message.

Describing a MIME multipart message

MIME multipart messages are described using a `mime:multipartRelated` element that contains a number of `mime:part` elements. To fully describe a MIME multipart message you must do the following:

1. Inside the `input` or `output` message you are sending as a MIME multipart message, add a `mime:mulipartRelated` element as the first child element of the enclosing message.
2. Add a `mime:part` child element to the `mime:multipartRelated` element and set its `name` attribute to a unique string.
3. Add a `soap:body` element as the child of the `mime:part` element and set its attributes appropriately.

   Tip

   If the contract had a default SOAP binding, you can copy the `soap:body` element from the corresponding message from the default binding into the MIME multipart message.

4. Add another `mime:part` child element to the `mime:multipartReleated` element and set its `name` attribute to a unique string.
5. Add a `mime:content` child element to the `mime:part` element to describe the contents of this part of the message.

   To fully describe the contents of a MIME message part the `mime:content` element has the following attributes:

Table 4.1. `mime:content` Attributes

Attribute	Description
`part`	Specifies the name of the WSDL message `part`, from the parent message definition, that is used as the content

Attribute	Description
	of this part of the MIME multipart message being placed on the wire.
`type`	The MIME type of the data in this message part. MIME types are defined as a type and a subtype using the syntax *`type/subtype`*. There are a number of predefined MIME types such as `image/jpeg` and `text/plain`. The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*[2] and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*[3].

6. For each additional MIME part, repeat steps Step 4 on page 38 and Step 5 on page 38.

Example

Example 4.2 on page 39 shows a WSDL fragment defining a service that stores X-rays in JPEG format. The image data, `xRay`, is stored as an xsd:base64binary and is packed into the MIME multipart message's second part, `imageData`. The remaining two parts of the input message, `patientName` and `patientNumber`, are sent in the first part of the MIME multipart image as part of the SOAP body.

Example 4.2. Contract using SOAP with Attachments

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="storRequest">
    <part name="patientName" type="xsd:string"/>
    <part name="patientNumber" type="xsd:int"/>
```

[2] ftp://ftp.isi.edu/in-notes/rfc2045.txt
[3] ftp://ftp.isi.edu/in-notes/rfc2046.txt

```
    <part name="xRay" type="xsd:base64Binary"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageBinding" type="tns:xRayStorage">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="store">
      <soap:operation soapAction="" style="document"/>
      <input name="storRequest">
        <mime:multipartRelated>
          <mime:part name="bodyPart">
            <soap:body use="literal"/>
          </mime:part>
          <mime:part name="imageData">
            <mime:content part="xRay" type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </input>
      <output name="storResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="xRayStorageService">
    <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
      <soap:address location="http://localhost:9000"/>
    </port>
  </service>
</definitions>
```

Chapter 5. Sending Binary Data with SOAP MTOM

SOAP Message Transmission Optimization Mechanism (MTOM) replaces SOAP with attachments as a mechanism for sending binary data as part of an XML message. Using MTOM with Fuse Services Framework requires adding the correct schema types to a service's contract and enabling the MTOM optimizations.

SOAP Message Transmission Optimization Mechanism (MTOM) specifies an optimized method for sending binary data as part of a SOAP message. Unlike SOAP with Attachments, MTOM requires the use of XML-binary Optimized Packaging (XOP) packages for transmitting binary data. Using MTOM to send binary data does not require you to fully define the MIME Multipart/Related message as part of the SOAP binding. It does, however, require that you do the following:

1. Annotate the data that you are going to send as an attachment.

   You can annotate either your WSDL or the Java class that implements your data.

2. Enable the runtime's MTOM support.

   This can be done either programmatically or through configuration.

3. Develop a `DataHandler` for the data being passed as an attachment.

Note

Developing `DataHandler`s is beyond the scope of this book.

Annotating Data Types to use MTOM

Overview

In WSDL, when defining a data type for passing along a block of binary data, such as an image file or a sound file, you define the element for the data to be of type xsd:base64Binary. By default, any element of type xsd:base64Binary results in the generation of a byte[] which can be serialized using MTOM. However, the default behavior of the code generators does not take full advantage of the serialization.

In order to fully take advantage of MTOM you must add annotations to either your service's WSDL document or the JAXB class that implements the binary data structure. Adding the annotations to the WSDL document forces the code generators to generate streaming data handlers for the binary data. Annotating the JAXB class involves specifying the proper content types and might also involve changing the type specification of the field containing the binary data.

WSDL first

Example 5.1 on page 42 shows a WSDL document for a Web service that uses a message which contains one string field, one integer field, and a binary field. The binary field is intended to carry a large image file, so it is not appropriate to send it as part of a normal SOAP message.

Example 5.1. Message for MTOM

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:xsd1="http://mediStor.org/types/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <schema targetNamespace="http://mediStor.org/types/"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary" />
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
```

```
    </schema>
  </types>

  <message name="storRequest">
    <part name="record" element="xsd1:xRay"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageSOAPBinding" type="tns:xRayStorage">
    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="store">
      <soap12:operation soapAction="" style="document"/>
      <input name="storRequest">
        <soap12:body use="literal"/>
      </input>
      <output name="storResponse">
        <soap12:body use="literal"/>
      </output>
    </operation>
  </binding>
  ...
</definitions>
```

If you want to use MTOM to send the binary part of the message as an optimized attachment you must add the `xmime:expectedContentTypes` attribute to the element containing the binary data. This attribute is defined in the `http://www.w3.org/2005/05/xmlmime` namespace and specifies the MIME types that the element is expected to contain. You can specify a comma separated list of MIME types. The setting of this attribute changes how the code generators create the JAXB class for the data. For most MIME types, the code generator creates a `DataHandler`. Some MIME types, such as those for images, have defined mappings.

Note

The MIME types are maintained by the Internet Assigned Numbers Authority(IANA) and are described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message*

Bodies[1] and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*[2].

Tip

For most uses you specify `application/octet-stream`.

Example 5.2 on page 44 shows how you can modify xRayType from Example 5.1 on page 42 for using MTOM.

Example 5.2. Binary Data for MTOM

```
...
  <types>
    <schema targetNamespace="http://mediStor.org/types/"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary"
                   xmime:expectedContentTypes="application/octet-stream"/>
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
    </schema>
  </types>
...
```

The generated JAXB class generated for xRayType no longer contains a byte[]. Instead the code generator sees the `xmime:expectedContentTypes` attribute and generates a `DataHandler` for the imageData field.

Note

You do not need to change the `binding` element to use MTOM. The runtime makes the appropriate changes when the data is sent.

Java first

If you are doing Java first development you can make your JAXB class MTOM ready by doing the following:

[1] ftp://ftp.isi.edu/in-notes/rfc2045.txt
[2] ftp://ftp.isi.edu/in-notes/rfc2046.txt

1. Make sure the field holding the binary data is a `DataHandler`.
2. Add the `@XmlMimeType()` annotation to the field containing the data you want to stream as an MTOM attachment.

Example 5.3 on page 45 shows a JAXB class annotated for using MTOM.

Example 5.3. JAXB Class for MTOM

```
@XmlType
public class XRayType {
    protected String patientName;
    protected int patientNumber;
    @XmlMimeType("application/octet-stream")
    protected DataHandler imageData;
  ...
}
```

Enabling MTOM

By default the Fuse Services Framework runtime does not enable MTOM support. It sends all binary data as either part of the normal SOAP message or as an unoptimized attachment. You can activate MTOM support either programmatically or through the use of configuration.

Using JAX-WS APIs

Overview

Both service providers and consumers must have the MTOM optimizations enabled. The JAX-WS APIs offer different mechanisms for each type of endpoint.

Service provider

If you published your service provider using the JAX-WS APIs you enable the runtime's MTOM support as follows:

1. Access the `Endpoint` object for your published service.

   The easiest way to access the `Endpoint` object is when you publish the endpoint. For more information see "Publishing a Service" in *Developing Applications Using JAX-WS*.

2. Get the SOAP binding from the `Endpoint` using its `getBinding()` method, as shown in Example 5.4 on page 47.

   Example 5.4. Getting the SOAP Binding from an Endpoint

   ```
   // Endpoint ep is declared previously
   SOAPBinding binding = (SOAPBinding)ep.getBinding();
   ```

   You must cast the returned binding object to a `SOAPBinding` object to access the MTOM property.

3. Set the binding's MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method, as shown in Example 5.5 on page 47.

   Example 5.5. Setting a Service Provider's MTOM Enabled Property

   ```
   binding.setMTOMEnabled(true);
   ```

Consumer

To MTOM enable a JAX-WS consumer you must do the following:

1. Cast the consumer's proxy to a `BindingProvider` object.

Tip

For information on getting a consumer proxy see "Developing a Consumer Without a WSDL Contract" in *Developing Applications Using JAX-WS* or "Developing a Consumer From a WSDL Contract" in *Developing Applications Using JAX-WS*.

2. Get the SOAP binding from the `BindingProvider` using its `getBinding()` method, as shown in Example 5.6 on page 48.

Example 5.6. Getting a SOAP Binding from a `BindingProvider`

```
// BindingProvider bp declared previously
SOAPBinding binding = (SOAPBinding)bp.getBinding();
```

3. Set the bindings MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method, as shown in Example 5.7 on page 48.

Example 5.7. Setting a Consumer's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

Using configuration

Overview

If you publish your service using XML, such as when deploying to a container, you can enable your endpoint's MTOM support in the endpoint's configuration file. For more information on configuring endpoint's see *Configuring Web Service Endpoints*.

Procedure

The MTOM property is set inside the `jaxws:endpoint` element for your endpoint. To enable MTOM do the following:

1. Add a `jaxws:property` child element to the endpoint's `jaxws:endpoint` element.
2. Add a `entry` child element to the `jaxws:property` element.
3. Set the `entry` element's `key` attribute to `mtom-enabled`.
4. Set the `entry` element's `value` attribute to `true`.

Example

Example 5.8 on page 49 shows an endpoint that is MTOM enabled.

Example 5.8. Configuration for Enabling MTOM

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

                            http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jax
ws.xsd">

  <jaxws:endpoint id="xRayStorage"
                  implementor="demo.spring.xRayStorImpl"
                  address="http://localhost/xRayStorage">
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>
</beans>
```

Chapter 6. Using XML Documents

The pure XML payload format provides an alternative to the SOAP binding by allowing services to exchange data using straight XML documents without the overhead of a SOAP envelope.

XML binding namespace

The extensions used to describe XML format bindings are defined in the namespace `http://cxf.apache.org/bindings/xformat`. Fuse Services Framework tools use the prefix xformat to represent the XML binding extensions. Add the following line to your contracts:

```
xmlns:xformat="http://cxf.apache.org/bindings/xformat"
```

Hand editing

To map an interface to a pure XML payload format do the following:

1. Add the namespace declaration to include the extensions defining the XML binding. See "XML binding namespace" on page 51.
2. Add a standard WSDL `binding` element to your contract to hold the XML binding, give the binding a unique `name`, and specify the name of the WSDL `portType` element that represents the interface being bound.
3. Add an `xformat:binding` child element to the `binding` element to identify that the messages are being handled as pure XML documents without SOAP envelopes.
4. Optionally, set the `xformat:binding` element's `rootNode` attribute to a valid QName. For more information on the effect of the `rootNode` attribute see "XML messages on the wire" on page 52.
5. For each operation defined in the bound interface, add a standard WSDL `operation` element to hold the binding information for the operation's messages.
6. For each operation added to the binding, add the `input`, `output`, and `fault` children elements to represent the messages used by the operation.

   These elements correspond to the messages defined in the interface definition of the logical operation.

7. Optionally add an `xformat:body` element with a valid `rootNode` attribute to the added `input`, `output`, and `fault` elements to override the value of `rootNode` set at the binding level.

Note

If any of your messages have no parts, for example the output message for an operation that returns void, you must set the `rootNode` attribute for the message to ensure that the message written on the wire is a valid, but empty, XML document.

XML messages on the wire

When you specify that an interface's messages are to be passed as XML documents, without a SOAP envelope, you must take care to ensure that your messages form valid XML documents when they are written on the wire. You also need to ensure that non-Fuse Services Framework participants that receive the XML documents understand the messages generated by Fuse Services Framework.

A simple way to solve both problems is to use the optional `rootNode` attribute on either the global `xformat:binding` element or on the individual message's `xformat:body` elements. The `rootNode` attribute specifies the QName for the element that serves as the root node for the XML document generated by Fuse Services Framework. When the `rootNode` attribute is not set, Fuse Services Framework uses the root element of the message part as the root element when using doc style messages, or an element using the message part name as the root element when using rpc style messages.

For example, if the `rootNode` attribute is not set the message defined in Example 6.1 on page 52 would generate an XML document with the root element `lineNumber`.

Example 6.1. Valid XML Binding Message

```
<type ... >
  ...
  <element name="operatorID" type="xsd:int"/>
  ...
</types>
<message name="operator">
  <part name="lineNumber" element="ns1:operatorID"/>
</message>
```

For messages with one part, Fuse Services Framework will always generate a valid XML document even if the `rootNode` attribute is not set. However,

the message in Example 6.2 on page 53 would generate an invalid XML document.

Example 6.2. Invalid XML Binding Message

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
  <element name="entryNum" type="xsd:int"/>
  ...
</types>

<message name="matildas">
  <part name="dancing" element="ns1:pairName"/>
  <part name="number" element="ns1:entryNum"/>
</message>
```

Without the `rootNode` attribute specified in the XML binding, Fuse Services Framework will generate an XML document similar to Example 6.3 on page 53 for the message defined in Example 6.2 on page 53. The generated XML document is invalid because it has two root elements: `pairName` and `entryNum`.

Example 6.3. Invalid XML Document

```
<pairName>
  Fred&Linda
</pairName>
<entryNum>
  123
</entryNum>
```

If you set the `rootNode` attribute, as shown in Example 6.4 on page 53 Fuse Services Framework will wrap the elements in the specified root element. In this example, the `rootNode` attribute is defined for the entire binding and specifies that the root element will be named entrants.

Example 6.4. XML Binding with rootNode set

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
  </operation>
</portType>

<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
```

```
    <input name="contestant"/>
    <output name="entered"/>
</binding>
```

An XML document generated from the input message would be similar to Example 6.5 on page 54. Notice that the XML document now only has one root element.

Example 6.5. XML Document generated using the rootNode attribute

```
<entrants>
  <pairName>
    Fred&Linda
  <entryNum>
    123
  </entryNum>
</entrants>
```

Overriding the binding's rootNode attribute setting

You can also set the `rootNode` attribute for each individual message, or override the global setting for a particular message, by using the `xformat:body` element inside of the message binding. For example, if you wanted the output message defined in Example 6.4 on page 53 to have a different root element from the input message, you could override the binding's root element as shown in Example 6.6 on page 54.

Example 6.6. Using `xformat:body`

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered">
      <xformat:body rootNode="entryStatus" />
    </output>
  </operation>
</binding>
```

Part II. Transports

Chapter 7. Understanding How Endpoints are Defined in WSDL

Endpoints represent an instantiated service. They are defined by combining a binding and the networking details used to expose the endpoint.

Overview

An endpoint can be thought of as a physical manifestation of a service. It combines a binding, which specifies the physical representation of the logical data used by a service, and a set of networking details that define the physical connection details used to make the service contactable by other endpoints.

Endpoints and services

In the same way a binding can only map a single interface, an endpoint can only map to a single service. However, a service can be manifested by any number of endpoints. For example, you could define a ticket selling service that was manifested by four different endpoints. However, you could not have a single endpoint that manifested both a ticket selling service and a widget selling service.

The WSDL elements

Endpoints are defined in a contract using a combination of the WSDL `service` element and the WSDL `port` element. The `service` element is a collection of related `port` elements. The `port` elements define the actual endpoints.

The WSDL `service` element has a single attribute, `name`, that specifies a unique name. The `service` element is used as the parent element of a collection of related `port` elements. WSDL makes no specification about how the `port` elements are related. You can associate the `port` elements in any manner you see fit.

The WSDL `port` element has a single attribute, `binding`, that specifies the binding used by the endpoint. The `port` element is the parent element of the elements that specify the actual transport details used by the endpoint. The elements used to specify the transport details are discussed in the following sections.

Adding endpoints to a contract

Fuse Services Framework provides command line tools that can generated endpoints for predefined service interface and binding combinations.

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different transports used in defining an endpoint work.

You can also add an endpoint to a contract using any text editor. When you hand edit a contract, you are responsible for ensuring that the contract is valid.

Supported transports

Endpoint definitions are built using extensions defined for each of the transports the Fuse Services Framework supports. This includes the following transports:

- HTTP
- CORBA
- Java Messaging Service

Chapter 8. Using HTTP

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is the assumed transport for most WS- specifications and is integral to RESTful architectures.*

Adding a Basic HTTP Endpoint

Overview

There are three ways of specifying an HTTP endpoint's address depending on the payload format you are using.

- SOAP 1.1 uses the standardized `soap:address` element.
- SOAP 1.2 uses the `soap12:address` element.
- All other payload formats use the `http:address element.`

SOAP 1.1

When you are sending SOAP 1.1 messages over HTTP you must use the SOAP 1.1 `address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL. The SOAP 1.1 `address` element is defined in the namespace `http://schemas.xmlsoap.org/wsdl/soap/`.

Example 8.1 on page 60 shows a `port` element used to send SOAP 1.1 messages over HTTP.

Example 8.1. SOAP 1.1 Port Element

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  ...
  <service name="SOAP11Service">
    <port binding="SOAP11Binding" name="SOAP11Port">
      <soap:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

SOAP 1.2

When you are sending SOAP 1.2 messages over HTTP you must use the SOAP 1.2 `address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL. The SOAP 1.2 `address` element is defined in the namespace `http://schemas.xmlsoap.org/wsdl/soap12/`.

Example 8.2 on page 61 shows a `port` element used to send SOAP 1.2 messages over HTTP.

Example 8.2. SOAP 1.2 Port Element

```
<definitions ...
             xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ... >
  <service name="SOAP12Service">
    <port binding="SOAP12Binding" name="SOAP12Port">
      <soap12:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

Other messages types

When your messages are mapped to any payload format other than SOAP you must use the HTTP `address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL. The HTTP `address` element is defined in the namespace `http://schemas.xmlsoap.org/wsdl/http/`.

Example 8.3 on page 61 shows a `port` element used to send an XML message.

Example 8.3. HTTP Port Element

```
<definitions ...
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ... >
  <service name="HTTPService">
    <port binding="HTTPBinding" name="HTTPPort">
      <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

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 8.4 on page 63 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 8.4. 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 8.5 on page 63 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 8.5. `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 8.1 on page 64.

Table 8.1. 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 64.
`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 8.2 on page 64, specify the connection's properties.

Table 8.2. 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 by the

Attribute	Description
	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. • `close`(default) — Specifies that the connection to the server is closed after each request/response sequence.

Attribute	Description
`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 70.
`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 83. 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.

Attribute	Description
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 8.6 on page 68 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 8.6. 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 B on page 121.

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 8.7 on page 69 to the `definitions` element of your endpoint's WSDL document.

Example 8.7. 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 8.2 on page 64.

Example

Example 8.8 on page 69 shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 8.8. WSDL to Configure an HTTP Consumer Endpoint

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

Consumer Cache Control Directives

Table 8.3 on page 70 lists the cache control directives supported by an HTTP consumer.

Table 8.3. `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 8.9 on page 72 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 8.9. 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 8.10 on page 72 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 8.10. `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 8.4 on page 73.

Table 8.4. 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 73.
`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 8.5 on page 73, specify the connection's properties.

Table 8.5. 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 set to `302` and the status description is set to `Object Moved`. The

Attribute	Description
	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 77.
`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 8.11 on page 74 shows the configuration for an HTTP service provider endpoint that honors keep-alive requests and suppresses all communication errors.

Example 8.11. 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 8.12 on page 76 to the `definitions` element of your endpoint's WSDL document.

Example 8.12. 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 8.5 on page 73.

Example

Example 8.13 on page 76 shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 8.13. 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 8.6 on page 77 lists the cache control directives supported by an HTTP service provider.

Table 8.6. `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 this directive, the proxy-revalidate directive must also be used.

Directive	Behavior
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 the Jetty Runtime

Overview

The Jetty runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Jetty runtime.

Namespace

The elements used to configure the Jetty runtime are defined in the namespace `http://cxf.apache.org/transports/http-jetty/configuration`. It is commonly referred to using the prefix httpj. In order to use the Jetty configuration elements you must add the lines shown in Example 8.14 on page 79 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 8.14. Jetty Runtime Configuration Namespace

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

The engine-factory element

The `httpj:engine-factory` element is the root element used to configure the Jetty runtime used by an application. It has a single required attribute, `bus`, whose value is the name of the `Bus` that manages the Jetty instances being configured.

Tip

The value is typically `cxf` which is the name of the default `Bus` instance.

The `httpj:engine-factory` element has three children that contain the information used to configure the HTTP ports instantiated by the Jetty runtime factory. The children are described in Table 8.7 on page 80.

Table 8.7. Elements for Configuring a Jetty Runtime Factory

Element	Description
`httpj:engine`	Specifies the configuration for a particular Jetty runtime instance. See "The `engine` element" on page 80.
`httpj:identifiedTLSServerParameters`	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, `id`, that specifies a unique identifier by which the property set can be referred.
`httpj:identifiedThreadingParameters`	Specifies a reusable set of properties for controlling a Jetty instance's thread pool. It has a single attribute, `id`, that specifies a unique identifier by which the property set can be referred. See "Configuring the thread pool" on page 81.

The engine element

The `httpj:engine` element is used to configure specific instances of the Jetty runtime. It has a single attribute, `port`, that specifies the number of the port being managed by the Jetty instance.

Tip

You can specify a value of `0` for the `port` attribute. Any threading properties specified in an `httpj:engine` element with its `port` attribute set to `0` are used as the configuration for all Jetty listeners that are not explicitly configured.

Each `httpj:engine` element can have two children: one for configuring security properties and one for configuring the Jetty instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent `httpj:engine-factory` element.

The child elements used to provide the configuration properties are described in Table 8.8 on page 80.

Table 8.8. Elements for Configuring a Jetty Runtime Instance

Element	Description
`httpj:tlsServerParameters`	Specifies a set of properties for configuring the security used for the specific Jetty instance.

Element	Description
`httpj:tlsServerParametersRef`	Refers to a set of security properties defined by a `identifiedTLSServerParameters` element. The `id` attribute provides the id of the referred `identifiedTLSServerParameters` element.
`httpj:threadingParameters`	Specifies the size of the thread pool used by the specific Jetty instance. See "Configuring the thread pool" on page 81.
`httpj:threadingParametersRef`	Refers to a set of properties defined by a `identifiedThreadingParameters` element. The `id` attribute provides the id of the referred `identifiedThreadingParameters` element.

Configuring the thread pool

You can configure the size of a Jetty instance's thread pool by either:

- Specifying the size of the thread pool using a `identifiedThreadingParameters` element in the `engine-factory` element. You then refer to the element using a `threadingParametersRef` element.
- Specifying the size of the of the thread pool directly using a `threadingParameters` element.

The `threadingParameters` has two attributes to specify the size of a thread pool. The attributes are described in Table 8.9 on page 81.

Note

The `httpj:identifiedThreadingParameters` element has a single child `threadingParameters` element.

Table 8.9. Attributes for Configuring a Jetty Thread Pool

Attribute	Description
`minThreads`	Specifies the minimum number of threads available to the Jetty instance for processing requests.

Attribute	Description
maxThreads	Specifies the maximum number of threads available to the Jetty instance for processing requests.

Example

Example 8.15 on page 82 shows a configuration fragment that configures a Jetty instance on port number 9001.

Example 8.15. Configuring a Jetty Instance

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
            http://cxf.apache.org/transports/http/configuration
            http://cxf.apache.org/schemas/configuration/http-conf.xsd
            http://cxf.apache.org/transports/http-jetty/configuration
            http://cxf.apache.org/schemas/configuration/http-jetty.xsd
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  ...

  <httpj:engine-factory bus="cxf">
    <httpj:identifiedTLSServerParameters id="secure">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                      file="certs/cherry.jks"/>
      </sec:keyManagers>
    </httpj:identifiedTLSServerParameters>

    <httpj:engine port="9001">
      <httpj:tlsServerParametersRef id="secure" />
      <httpj:threadingParameters minThreads="5"
                                 maxThreads="15" />
    </httpj:engine>
  </httpj:engine-factory>
</beans>
```

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 83.

2. Configure the consumer to use a decoupled endpoint.

   See "Configuring the consumer" on page 84.

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

   See "Configuring an endpoint to use WS-Addressing" on page 83.

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 8.16 on page 84.

Example 8.16. 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 8.17 on page 84.

Example 8.17. 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 8.18 on page 85 shows the configuration for setting up the endpoint defined in Example 8.16 on page 84 to use use a decoupled endpoint. The consumer now receives all responses at `http://widgetvendor.net/widgetSellerInbox`.

Example 8.18. Configuring a Consumer to Use a Decoupled HTTP Endpoint

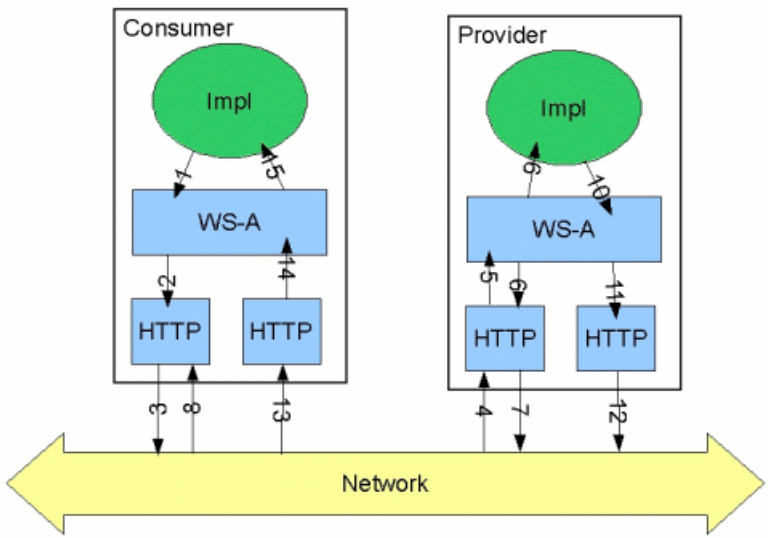
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           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 8.1 on page 86 shows the flow of messages when using HTTP in decoupled mode.

Figure 8.1. 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 9. 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 9.1 on page 90 shows a WSDL binding that uses SOAP/JMS.

Example 9.1. 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 SOAP/HTTP endpoint. The difference is the address specification. JMS endpoints use a

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

JMS URI as defined in the URI Scheme for JMS 1.0[2].
Example 9.2 on page 91 shows the syntax for a JMS URI.

Example 9.2. JMS URI syntax

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

Table 9.1 on page 91 describes the available variants for the JMS URI.

Table 9.1. 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 93.

Example 9.3 on page 91 shows the WSDL port entry for a SOAP/JMS endpoint whose target destination is looked up using JNDI.

Example 9.3. SOAP/JMS endpoint address

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

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

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

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 9.2 on page 91, 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 9.4 on page 93 shows the syntax for using multiple options in a JMS URI.

Example 9.4. Syntax for JMS URI options

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

JMS properties

Table 9.2 on page 93 shows the URI options that affect the JMS transport layer.

Table 9.2. 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

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

Property	Default	Description
		applications that have request-reply 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 9.3 on page 94 shows the URI options that can be used to configure JNDI for this endpoint.

Table 9.3. 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 9.5 on page 95.

Example 9.5. 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 9.3 on page 94). 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 9.6 on page 95.

Example 9.6. 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 9.4 on page 96 shows all of the WSDL extension elements you can use to configure the JMS transport.

Table 9.4. 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.

Element	Default	Description
soapjms:jndiConnectionFactoryName		Specifies the JNDI name of the JMS connection factory.
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

Element	Default	Description
		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 9.7 on page 98 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 9.7. 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 9.7 on page 98 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 10. 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.

Configuration namespace

The JMS configuration bean uses the Spring p-namespace[1] to make the configuration as simple as possible. To use this namespace you need to declare it in the configuration's root element as shown in Example 10.1 on page 102.

Example 10.1. Declaring the Spring p-namespace

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

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 10.1 on page 102 lists properties that are common to both providers and consumers.

Table 10.1. 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

[1] http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-p-namespace

Property	Default	Description
		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. For more details see "Using a Named Reply Destination" on page 116.
`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

Property	Default	Description
		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.
`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).

Property	Default	Description
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
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.
messageSelector		Specifies the string value of the selector. For more information on the syntax used to specify message

Property	Default	Description
		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.
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 10.2 on page 106, the bean's properties are specified as attributes to the bean element. They are all declared in the Spring p namespace.

Example 10.2. 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 10.3 on page 107 shows a JAX-WS client that uses the JMS configuration from Example 10.2 on page 106.

Example 10.3. 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 10.4 on page 107 shows a JMS conduit that uses the JMS configuration from Example 10.2 on page 106.

Example 10.4. 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 10.5 on page 108 to the definitions element of your contract.

Example 10.5. 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 10.2 on page 109.

Table 10.2. 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 116.
`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. For more details see "Using a Named Reply Destination" on page 116.

Attribute	Description
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 10.6 on page 111 shows an example of a JMS WSDL `port` specification.

Example 10.6. 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 10.3. 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 10.7 on page 113 shows the WSDL for configuring a JMS consumer endpoint.

Example 10.7. 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 10.4. 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 10.8 on page 115 shows the WSDL for configuring a JMS provider endpoint.

Example 10.8. 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 10.9 on page 116 shows the configuration for a JMS client endpoint.

Example 10.9. 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>
```

Part III. Appendices

Appendix A. Integrating with Fuse Message Broker

Overview

If you are using Fuse Message Broker or Apache ActiveMQ as your JMS provider, the JNDI name of your destinations can be specified in a special format that dynamically creates JNDI bindings for queues or topics. This means that it is *not* necessary to configure the JMS provider in advance with the JNDI bindings for your queues or topics.

The initial context factory

The key to integrating Fuse Message Broker or with JNDI is the `ActiveMQInitialContextFactory` class. This class is used to create a JNDI `InitialContext` instance, which you can then use to access JMS destinations in the JMS broker.

Example A.1 on page 119 shows SOAP/JMS WSDL extensions to create a JNDI `InitialContext` that is integrated with Fuse Message Broker.

Example A.1. SOAP/JMS WSDL to connect to Fuse Message Broker

```
<soapjms:jndiInitialContextFactory>
  org.apache.activemq.jndi.ActiveMQInitialContextFactory
</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
```

In Example A.1 on page 119, the Fuse Message Broker client connects to the broker port located at `tcp://localhost:61616`.

Looking up the connection factory

As well as creating a JNDI `InitialContext` instance, you must specify the JNDI name that is bound to a `javax.jms.ConnectionFactory` instance. In the case of Fuse Message Broker and Apache ActiveMQ, there is a predefined binding in the `InitialContext` instance, which maps the JNDI name `ConnectionFactory` to an `ActiveMQConnectionFactory` instance. Example A.2 on page 120 shaows the SOAP/JMS extension element for specifying the Fuse Message Broker connection factory.

Example A.2. SOAP/JMS WSDL for specifying the Fuse Message Broker connection factory

```
<soapjms:jndiConnectionFactoryName>
  ConnectionFactory
</soapjms:jndiConnectionFactoryName>
```

Syntax for dynamic destinations

To access queues or topics dynamically, specify the destination's JNDI name as a JNDI composite name in either of the following formats:

```
dynamicQueues/QueueName
dynamicTopics/TopicName
```

QueueName and *TopicName* are the names that the Apache ActiveMQ broker uses. They are *not* abstract JNDI names.

Example A.3 on page 120 shows a WSDL port that uses a dynamically created queue.

Example A.3. WSDL port specification with a dynamically created queue

```
<service name="JMSService">
  <port binding="tns:GreeterBinding" name="JMSPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/greeter.request.queue" >
      <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>
```

When the application attempts to open the JMS connection, Fuse Message Broker will check to see if a queue with the JNDI name `greeter.request.queue` exists. If it does not exist, it will create a new queue and bind it to the JNDI name `greeter.request.queue`.

Appendix B. 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.

Appendix C. Conduit and Destination Based JMS Configuration

Versions of Fuse Services Framework previous to 2.1.3 configured the JMS transport at the conduit and destination level. This configuration can still be used, but may be removed in future versions.

The Fuse Services Framework JMS configuration properties are specified under the `http://cxf.apache.org/transports/jms` namespace. In order to use the JMS configuration properties you will need to add the line shown in Example C.1 on page 123 to the beans element of your configuration.

Example C.1. JMS Configuration Namespaces

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

Basic endpoint configuration

Overview

JMS endpoints are configured using Spring configuration. You can configure the server-side and consumer-side transports independently.

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.

Configuration elements

You configure a JMS endpoint using one of the following configuration elements:

`jms:conduit`

The `jms:conduit` element contains the configuration for a consumer endpoint. It has one attribute, `name`, whose value takes the form `{`*`WSDLNamespace`*`}`*`WSDLPortName`*`.jms-conduit`.

`jms:destination`

The `jms:destination` element contains the configuration for a provider endpoint. It has one attribute, `name`, whose value takes the form `{`*`WSDLNamespace`*`}`*`WSDLPortName`*`.jms-destination`.

The address element

JMS connection information is specified by adding a `jms:address` child to the base configuration element. The `jms:address` element uses the attributes described in Table 10.2 on page 109 to configure the connection to the JMS broker.

The JMSNamingProperties element

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 C.2 on page 126 shows a Fuse Services Framework configuration entry for configuring the addressing information for a JMS consumer endpoint.

Example C.2. Addressing Information in a Fuse Services Framework Configuration File

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ct="http://cxf.apache.org/configuration/types"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                          http://cxf.apache.org/jaxws
                           http://cxf.apache.org/schemas/jaxws.xsd
                          http://cxf.apache.org/transports/jms
                           http://cxf.apache.org/schemas/configuration/jms.xsd">
  <jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
                 jndiConnectionFactoryName="myConnectionFactory"
                 jndiDestinationName="myDestination"
                 jndiReplyDestinationName="myReplyDestination"
                 connectionUserName="testUser"
                 connectionPassword="testPassword">
      <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>
</beans>
```

Consumer configuration

Overview

The JMS consumer configuration allows you to specify the following:

- the type of message.
- the number of milliseconds the consumer will wait for a response.
- the number of milliseconds a request will exist before the JMS broker can remove it.

Configuration elements

Consumer endpoint configuration is specified using the `jms:conduit` element. It has two children:

- `jms:runtimePolicy`—configures the JMS message type
- `jms:clientConfig`—configures the response timeout and the message's time to live

---

Specifying the message type

The `jms:runtimePolicy` element specifies the message type supported by the consumer endpoint using . The message type is specified using the `messageType` attribute. The `messageType` attribute has two possible values:

Table C.1. messageType values

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

---

jms:clientConfig

The `jms:clientConfig` element two attributes that are used to specify the configurable runtime properties of a consumer endpoint:

`clientReceiveTimeout`

The `clientReceiveTimeout` attribute specifies the interval, in milliseconds, a consumer endpoint will wait for a response before timing out. The default timeout interval is 2000.

`messageTimeToLive`

The `messageTimeToLive` attribute specifies the interval, in milliseconds, that a request can remain unreceived before the JMS broker can delete it. The default time to live interval is 0 which specifies that the request has an infinite time to live.

Example

Example C.3 on page 128 shows a configuration entry for configuring a JMS consumer endpoint.

Example C.3. Configuration for a JMS Consumer Endpoint

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ct="http://cxf.apache.org/configuration/types"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                          http://cxf.apache.org/jaxws
                           http://cxf.apache.org/schemas/jaxws.xsd
                          http://cxf.apache.org/transports/jms
                           http://cxf.apache.org/schemas/configuration/jms.xsd">
   ...
  <jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address ... >
                    ...
    </jms:address>
                    ...
    <jms:runtimePolicy messageType="binary"/>
    <jms:clientConfig clientReceiveTimeout="500"
                      messageTimeToLive="500" />
  </jms:conduit>
  ...
</beans>
```

Provider configuration

Overview

The provider specific configuration allows you to specify two types of properties:

- JMS messaging qualities of service
- JMS runtime behaviors

Configuration elements

Provider endpoint configuration is specified using the `jms:destination` element. It has two children:

- `jms:runtimePolicy`—configures the JMS qualities of service
- `jms:serverConfig`—configures the runtime behavior

Quality of service configuration

The `jms:runtimePolicy` element specifies the qualities of service for the provider endpoint. Table C.2 on page 129 describes the attributes used to configure the qualities of service.

Table C.2. Provider Endpoint Configuration

Attribute	Description
`useMessageIDAsCorrealationID`	Specifies whether the JMS broker 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.

Attribute	Description
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.

Runtime configuration

You configure provider runtime behavior using the jms:serverConfig element. It has two attributes that are used to specify the configurable runtime properties of a provider endpoint:

messageTimeToLive

The messageTimeToLive attribute specifies the amount of time, in milliseconds, that a response can remain unread before the JMS broker is allowed to delete it. The default is 0 which specifies that the message can live forever.

durableSubscriptionClientId

The durableSubscriptionClientId attribute specifies the client identifier the endpoint uses to create and access durable subscriptions.

Example

Example C.4 on page 130 shows a Fuse Services Framework configuration entry for configuring a provider endpoint.

Example C.4. Configuration for a Provider Endpoint

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ct="http://cxf.apache.org/configuration/types"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                         http://cxf.apache.org/jaxws
                           http://cxf.apache.org/schemas/jaxws.xsd
                         http://cxf.apache.org/transports/jms
                           http://cxf.apache.org/schemas/configuration/jms.xsd">
   ...
  <jms:destination name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-destination">

     ...
```

```
    <jms:runtimePolicy messageSelector="cxf_message_selector"
                       useMessageIDAsCorrelationID="true"
                       transactional="true"
                       durableSubscriberName="cxf_subscriber" />
  <jms:serverConfig messageTimeToLive="500"
                    durableSubscriptionClientId="jms-test-id" />
  </jms:destination>
  ...
</beans>
```

JMS Session Pool Configuration

Overview

The JMS configuration allows you to specify the number of JMS sessions an endpoint will keep in a pool.

Configuration element

You use the `jms:sessionPool` element to specify the session pool configuration for a JMS endpoint. The `jms:sessionPool` element is a child of both the `jms:conduit` element and the `jms:destination` element.

The `jms:sessionPool` element's attributes, listed in Table C.3 on page 132, specify the high and low water marks for the endpoint's JMS session pool.

Table C.3. Attributes for Configuring the JMS Session Pool

Attribute	Description
`lowWaterMark`	Specifies the minimum number of JMS sessions pooled by the endpoint. The default is `20`.
`highWaterMark`	Specifies the maximum number of JMS sessions pooled by the endpoint. The default is `500`.

Example

Example C.5 on page 132 shows an example of configuring the session pool for a Fuse Services Framework JMS provider endpoint.

Example C.5. JMS Session Pool Configuration

```
...
<jms:destination name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-destination">

  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:sessionPool lowWaterMark="10"
                   highWaterMark="5000" />
  ...
</jms:destination>
...
```

Index

J

M

N

P

S

W

X