Artix ESB

Bindings and Transports, Java Runtime

Version 5.0
July 2007

Making Software Work Together™

Bindings and Transports, Java Runtime

IONA Technologies

Version 5.0

Published 24 Jul 2007

Trademark and Disclaimer Notice

Copyright Notice

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

What is Covered in This Book

This book discusses the bindings and transports supported by the Artix ESB Java Runtime. It describes how the combination of WSDL elements and configuration is used to set-up a binding or a transport. It also discusses the advantages of using each of the bindings and transports.

Who Should Read This Book

This book is intended for people who are developing the contracts for endpoints that are going to be deployed into the Artix ESB Java Runtime. It assumes a working knowledge of WSDL and XML. It also assumes a working knowledge of the underlying middleware technology being discussed.

How to Use This Book

This book is broken into two parts:

- Part I, “Bindings” describes how to work with the message bindings.
- Part II, “Transports” describes how to work with the transports.

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see Using the Artix ESB Librabry [../library_intro/index.htm].

Part I. Bindings

Table of Contents

Chapter 1. Understanding Bindings in WSDL

Summary

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 “Understanding How Endpoints are Defined WSDL” on page 195.

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

Artix provides a number of tools for adding bindings to your contracts. These include:

- Artix Designer has wizards that lead you through the process of adding bindings to your contract.
- A number of the bindings can be generated using command line tools.

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 you hand edit a contract, you are responsible for ensuring that the contract is valid.

Supported bindings

The Artix ESB Java Runtime supports the following bindings:

- SOAP 1.1
- SOAP 1.2
- CORBA
- Pure XML

Chapter 2. Using SOAP 1.1 Messages

Summary

Artix ESB 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

Using the command line

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

`artix 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 tool has the following required arguments:

Option	Interpretation
-i *`port-type-name`*	Specifies the `portType` element for which a binding should be 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 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.

Important

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

Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in a WSDL fragment similar to the one shown in Example 2.1, “Ordering System Interface”.

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, "SOAP 1.1 Binding for `orderWidgets`".

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, “SOAP Header Syntax”. 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, Artix ESB automatically fills in the SOAP headers for you.

Example

Example 2.4, "SOAP 1.1 Binding with a SOAP Header" shows a modified version of the `orderWidgets` service shown in Example 2.1, "Ordering System Interface". 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 would be responsible for adding the SOAP header in 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 could modify Example 2.4, “SOAP 1.1 Binding with a SOAP Header” so that the header value was a part of the input and output messages as shown in Example 2.5, “SOAP 1.1 Binding for orderWidgets with a SOAP Header”. In this case `keyVal` is a part of the input and output messages. In the `soap:body` element's `parts` attribute specifies that `keyVal` is not to 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

Summary

Artix ESB 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 the command line

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

`artix 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 should be 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 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.

Important

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

Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in a WSDL fragment similar to the one shown in Example 3.1, “Ordering System Interface”.

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, "SOAP 1.2 Binding for orderWidgets".

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 header becomes part of the parent message. A 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, “SOAP Header Syntax”.

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, “`soap12:header` Attributes”.

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 whether 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`, then the 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 want to 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, Artix ESB automatically fills in the SOAP headers for you.

Example

Example 3.4, “SOAP 1.2 Binding with a SOAP Header” shows a modified version of the `orderWidgets` service shown in Example 3.1, “Ordering System Interface”. This version has been modified so that each order has an xsd:base64binary value placed in the header of the request and response. The header is defined as being the `keyVal` part from the `widgetKey` message. In this case you would be 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 could modify Example 3.4, “SOAP 1.2 Binding with a SOAP Header” so that the header value was a part of the input and output messages as shown in Example 3.5, “SOAP 1.2 Binding for orderWidgets with a SOAP Header”. In this case `keyVal` is a part of the input and output messages. In the `soap12:body` elements the `parts` attribute specifies that `keyVal` is not to 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

Summary

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* (http://www.w3.org/TR/SOAP-attachments).

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, "MIME Namespace Specification in a Contract".

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 Artix ESB that the message body is going to be a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their contents. `mime:multipartRelated` elements in 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 do the following:

1. Inside the `input` or `output` message you want to send 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 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*(ftp://ftp.isi.edu/in-notes/rfc2045.txt) and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Type*s(ftp://ftp.isi.edu/in-notes/rfc2046.txt).

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

Example

Example 4.2, “Contract using SOAP with Attachments” 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"/>
    <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

Summary

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 Artix ESB 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

When defining a data type for passing along a block of binary data, such as an image file or a sound file, in WSDL 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 may also involve changing the type specification of the field containing the binary data.

WSDL first

Example 5.1, “Message for MTOM” 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 for sending along 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 wanted to use MTOM to send the binary part of the message as an optimized attachment you would need to 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 will change how the code generators create the JAXB class for the data. For most MIME types, the code generator will create 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 described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*(ftp://ftp.isi.edu/in-notes/rfc2045.txt) and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*(ftp://ftp.isi.edu/in-notes/rfc2046.txt)

Tip

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

Example 5.2, “Binary Data for MTOM” shows how you would modify xRayType from Example 5.1, “Message for MTOM” 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 will no longer contain a byte[]. Instead the code generator will see the `xmime:expectedContentTypes` attribute and generate a `DataHandler` for the imageData field.

Note

You do not need to change the `binding` element to use MTOM. The runtime will make 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. 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 be streamed as an MTOM attachment.

Example 5.3, “JAXB Class for MTOM” 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 Artix ESB runtime does not enable MTOM support. It will send 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

Both service providers and consumers need to 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. Get access to the `Endpoint` object for your published service.

   The easiest way to get the `Endpoint` object is when you publish the endpoint. For more information see Chapter 3, Publishing a Service in *Developing Artix Applications with JAX-WS*.

2. Get the SOAP binding from the `Endpoint` using its `getBinding()` method as shown in Example 5.4, “Getting the SOAP Binding from an Endpoint”.

   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 inorder to access the MTOM property.

3. Set the bindings MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method as shown in Example 5.5, “Setting a Service Provider's MTOM Enabled Property”.

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

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

Consumer

To MTOM enable a JAX-WS consumer you 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 Artix Applications with JAX-WS* or Developing a Consumer Starting from a WSDL Contract in *Developing Artix Applications with JAX-WS*.

2. Get the SOAP binding from the `BindingProvider` using its `getBinding()` method as shown in Example 5.6, “Getting a SOAP Binding from a `BindingProvider`”.

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, “Setting a Consumer's MTOM Enabled Property”.

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 into 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 and Deploying Artix Java Runtime 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, “Configuration for Enabling MTOM” 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/jaxws.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

Summary

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.

Artix Designer provides a wizard for generating an XML binding from a logical interface. Alternatively, you can create an XML binding using any text or XML editor.

XML binding namespace

The extensions used to describe XML format bindings are defined in the namespace `http://cxf.apache.org/bindings/xformat`. Artix ESB 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"
```

Using Artix Designer

You can add an XML binding to a contract by either selecting Artix Designer → New Binding or selecting New Binding from the context menu available in Artix Designer's diagram view. For more information see the on-line help provided with Artix Designer.

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.
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.
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-Artix participants that receive the XML documents understand the messages generated by Artix.

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 Artix ESB. When the `rootNode` attribute is not set, Artix ESB 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, "Valid XML Binding Message" 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, Artix will always generate a valid XML document even if the `rootNode` attribute is not set. However, the message in Example 6.2, “Invalid XML Binding Message” 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, Artix will generate an XML document similar to Example 6.3, “Invalid XML Document” for the message defined in Example 6.2, “Invalid XML Binding Message”. 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, “XML Binding with rootNode set” Artix ESB 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, “XML Document generated using the rootNode attribute”. 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, “XML Binding with rootNode set” to have a different root element from the input message, you could override the binding's root element as shown in Example 6.6, “Using `xformat:body`”.

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

Table of Contents

Chapter 7. Understanding How Endpoints are Defined in WSDL

Summary

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

Artix provides a number of tools for adding endpoints to your contracts. These include:

- Artix Designer has wizards that lead you through the process of adding endpoints to your contract.
- A number of the endpoint types can be generated using command line tools.

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 Artix ESB Java Runtime supports. This includes the following transports:

- HTTP
- IBM WebSphere MQ
- CORBA
- Java Messaging Service
- File Transfer Protocol

Chapter 8. Using HTTP

Summary

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Becuase 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, "SOAP 1.1 port Element" 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, "SOAP 1.2 port Element" 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, "HTTP port Element" 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 refered to using the prefix http-conf. In order to use the HTTP configuration elements you will need to add the lines shown in Example 8.4, “HTTP Consumer Configuration Namespace” to the `beans` element of your endpoint's configuration file. In addition, you will need to 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 endpoint using the `http-conf:conduit` element and its children. The `http-conf:conduit` element takes a single attribute, `id`, that specifies the WSDL `port` element that corresponds to the endpoint. The value for the `id` attribute takes the form *`portQName`*`.http-conduit`. For example, Example 8.5, "`http-conf:conduit` Element" shows the `http-conf:conduit` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> if the endpoint's target namespace was `http://widgets.widgetvendor.net`.

Example 8.5. `http-conf:conduit` Element

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

    ...
  </http-conf:conduit>
...
```

The `http-conf:conduit` element has a number of child elements that specify configuration information. They are described in Table 8.1, "Elements Used to Configure an HTTP Consumer Endpoint".

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.
`http-conf:authorization`	Specifies the the parameters for configuring the basic authentication method that the endpoint uses preemptively.  **Tip** 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.

Element	Description
`http-conf:basicAuthSupplier`	Specifies the bean reference or class name of the object that supplies the the basic authentication information used by the endpoint both 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 in order 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, “HTTP Consumer Configuration Attributes”, 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 will attempt to establish a connection before it times out. The default is `30000`. `0` specifies that the consumer will continue to send the request indefinitely.
`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.

Attribute	Description
	 **Important** Chunking cannot be used 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 as multipurpose internet mail extensions (MIME) types.
`AcceptLanguage`	Specifies what language (for example, American English) the consumer prefers for the purposes 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 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`.  **Tip** 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`.

Attribute	Description
`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.  **Tip** 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 to keep its connection 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.
`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.
`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 upon 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.  **Note** 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.

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

Example

Example 8.6, "HTTP Consumer Endpoint Configuration" shows a the configuration for an HTTP consumer endpoint that wants to keep its connection to the provider open between requests, will only retransmit requests once per invocation, and 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>
```

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 refered to using the prefix http-conf. In order to use the HTTP configuration elements you will need to add the line shown in Example 8.7, "HTTP Consumer WSDL Element's Namespace" 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, "HTTP Consumer Configuration Attributes".

Example

Example 8.8, "WSDL to Configure an HTTP Consumer Endpoint" shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it will 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, "`http-conf:client` Cache Control Directives" 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 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, it means the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that will be still be 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 might 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 at least 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 refered to using the prefix http-conf. In order to use the HTTP configuration elements you will need to add the lines shown in Example 8.9, "HTTP Provider Configuration Namespace" to the `beans` element of your endpoint's configuration file. In addition, you will need to 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, `id`, the specifies the WSDL `port` element that corresponds to the endpoint. The value for the `id` attribute takes the form *`portQName`*`.http-destination`. For example, Example 8.10, "`http-conf:destination` Element"

shows the `http-conf:destination` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> if the endpoint's target namespace was `http://widgets.widgetvendor.net`.

Example 8.10. `http-conf:destination` Element

```
...
  <http-conf:destination
id="{http://widgets/widgetvendor.net}widgetSOAPPort.http-destination>
    ...
  </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, “Elements Used to Configure an HTTP Service Provider Endpoint”.

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.
`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, “HTTP Service Provider Configuration Attributes”, 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 tries to receive a request before the connection times out. The default is `30000`.

Attribute	Description
	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 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.
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.

Attribute	Description
	 **Note** The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Artix 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, "HTTP Service Provider Endpoint Configuration" shows a 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-destination">
    <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 refered to using the prefix http-conf. In order to use the HTTP configuration elements you will need to add the line shown in Example 8.12, "HTTP Provider WSDL Element's Namespace" 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, "HTTP Service Provider Configuration Attributes".

Example

Example 8.13, "WSDL to Configure an HTTP Service Provider Endpoint" shows a WSDL fragment that configures an HTTP service provider endpoint to specify 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, "`http-conf:server` Cache Control Directives" 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 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 revaildate expired entries that relate to a response before that entry can be used in a subsequent response.
proxy-revalidate	Means the same as must-revalidate, except that it can only be enforced on shared caches and is ignored by private unshared caches. If 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	Means 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. If using this directive, the proxy-revalidate directive must also be used.
cache-extension	Specifies additional extensions to the other cache directives. Extensions might 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 at least 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. 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 refered to using the prefix httpj. In order to use the Jetty configuration elements you will need to add the lines shown in Example 8.14, "Jetty Runtime Configuration Namespace" to the `beans` element of your endpoint's configuration file. In addition, you will need to 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, "Elements for Configuring a Jetty Runtime Factory".

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.

Element	Description
`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 refered.
`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 refered. See Configuring the thread pool.

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 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, “Elements for Configuring a Jetty Runtime Instance”.

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 refered `identifiedTLSServerParameters` element.
`httpj:threadingParameters`	Specifies the size of the thread pool used by the specific Jetty instance. See Configuring the thread pool.
`httpj:threadingParametersRef`	Refers to a set of properties defined by a `identifiedThreadingParameters` element. The `id` attribute provides the id of the refered `identifiedThreadingParameters` element.

Configuring the thread pool

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

- Specifying the size of thread pool using a `identifiedThreadingParameters` element in the `engine-factory` element. You then refer to the element using a `threadingParametersRef` element.
- Specify the size of the of 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, “Attributes for Configuring a Jetty Thread Pool”.

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

Example

Example 8.15, “Configuring a Jetty Instance” 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 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 mode

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

1. Specify that the consumer and any service provider with which the consumer will interact 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, “Activating WS-Addressing using WSDL”.

   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, "Activating WS-Addressing using a Policy".

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.

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

   Example 8.18, "Configuring a Consumer to Use a Decoupled HTTP Endpoint" shows the configuration for the setting up the endpoint defined in Example 8.16, "Activating WS-Addressing using WSDL" to use use a decoupled endpoint. The consumer will now receive 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">
```

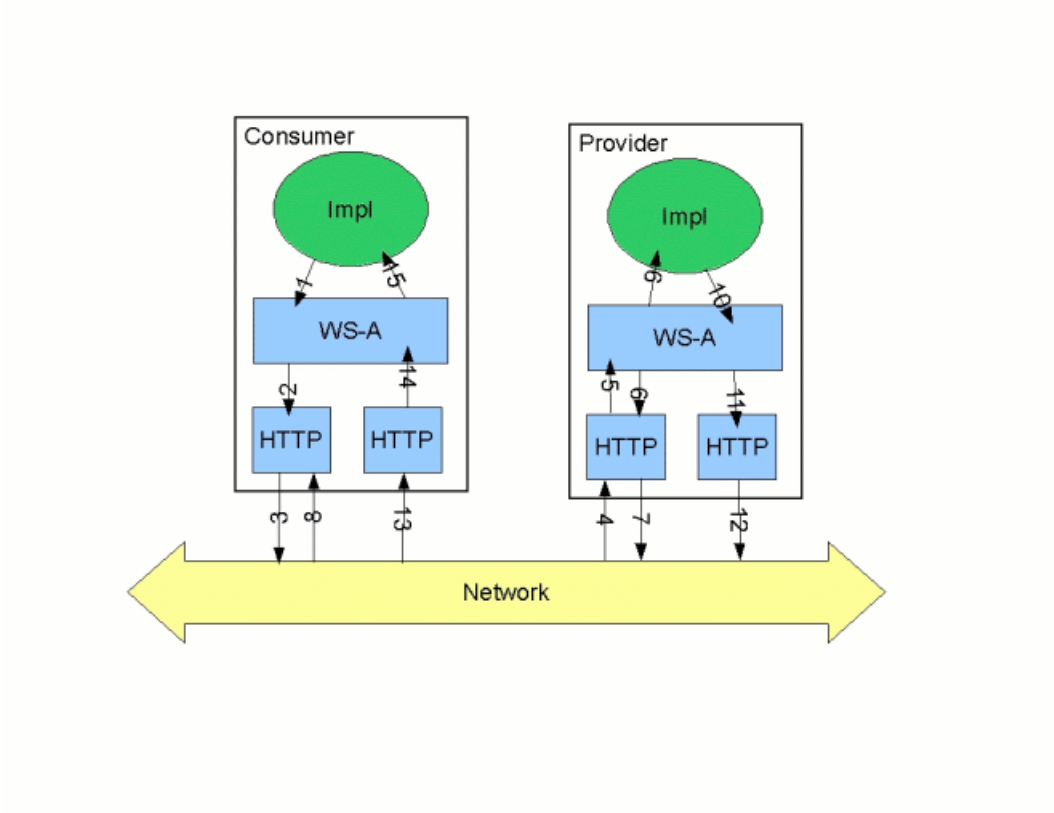
```
  <http:conduit name="{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">

    <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 may be important to understand what happens for debugging reasons.

Figure 8.1, “Message Flow in for a Decoupled HTTP Transport” 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 as far as 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` to acknowledge 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 is closed.
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 the JMS

Summary

The JMS is a standards based messaging system that is widely used in enterprise Java applications.

Namespaces

WSDL Namespace

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 9.1, "JMS Extension Namespace" to the definitions element of your contract.

Example 9.1. JMS Extension Namespace

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

Configuration Namespace

The Artix ESB JMS endpoint 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 9.2, "JMS Configuration Namespaces" to the beans element of your configuration.

Example 9.2. JMS Configuration Namespaces

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

Basic Endpoint Configuration

JMS endpoints need to know certain basic information about how to establish a connection to the proper destination. This information can be provided in one of two places:

- WSDL
- Configuration

Using WSDL

The JMS destination 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.

The `address` element

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 uses the attributes described in Table 9.1, "JMS Endpoint Attributes" to configure the connection to the JMS broker.

Table 9.1. 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.
`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.
`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 `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.

Using a named reply destination

By default, Artix ESB endpoints using JMS create a temporary queue for sending replies back and forth. You can change this behavior by setting the `jndiReplyDestinationName` attribute in the endpoint's contract.

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 9.3, "JMS WSDL Port Specification" shows an example of a JMS WSDL `port` specification.

Example 9.3. 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>
```

Using Configuration

In addition to using the WSDL file to specify the connection information for a JMS endpoint, you can supply it in the endpoint's configuration file. The information in the configuration file will override the information in the endpoint's WSDL file.

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 used in the configuration file is identical to the one used in the WSDL file. Its attributes are listed in Table 9.1, "JMS Endpoint Attributes". Like the `jms:address` element in the WSDL file, the `jms:address` configuration element also has a `jms:JMSNamingProperties` child element that is used to specify additional information used to connect to a JNDI provider.

Example

Example 9.4, "Addressing Information in a Artix ESB Configuration File" shows a Artix ESB configuration entry for configuring the addressing information for a JMS consumer endpoint.

Example 9.4. Addressing Information in a Artix ESB 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">
<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 Endpoint Configuration

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS `ObjectMessage` or a JMS `TextMessage`. When using an `ObjectMessage` 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 Artix ESB consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the Artix ESB 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.

A consumer endpoint can be configured in one of two ways:

- Configuration
- WSDL

Tip

The recommended method is to place the consumer endpoint specific information into the Artix ESB configuration file for the endpoint.

Using Configuration

Specifying the message type

Consumer endpoint configuration is specified using the `jms:conduit` element. Using this configuration element, you specify the message type supported by the consumer endpoint using the `jms:runtimePolicy` child element. The message type is specified using the `messageType` attribute. The `messageType` attribute has two possible values:

Table 9.2. `messageType` Values

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

Example

Example 9.5, “Configuration for a JMS Consumer Endpoint” shows a configuration entry for configuring a JMS consumer endpoint.

Example 9.5. 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">
...
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:runtimePolicy messageType="binary"/>
  ...
</jms:conduit>
...
</beans>
```

Using WSDL

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 9.3. 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 `ObjectMessage`.

Provider Endpoint Configuration

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

Service endpoints can be configure in one of two ways:

- Configuration
- WSDL

Tip

The recommended method is to place the provider endpoint specific information into the Artix ESB configuration file for the endpoint.

Using Configuration

Specifying configuration data

Provider endpoint configuration is specified using the `jms:destination` configuration element. Using this configuration element, you can specify the provider endpoint's behaviors using the `jms:runtimePolicy` element. When configuring a provider endpoint you can use the following `jms:runtimePolicy` attributes:

Table 9.4. 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.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false`. Currently, this is not supported by the runtime.

Example

Example 9.6, “Configuration for a Provider Endpoint” shows a Artix ESB configuration entry for configuring a provider endpoint.

Example 9.6. 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">
...
<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:destination>
...
</beans>
```

Using WSDL

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 9.5. 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.

Attribute	Description
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false`. Currently, this is not supported by the runtime.

JMS Runtime Configuration

In addition to configuring the externally visible aspects of your JMS endpoint, you can also configure aspects of its internal runtime behavior. There are three types of runtime configuration:

- JMS session pool configuration
- Consumer specific configuration
- Provider specific configuration

JMS Session Pool Configuration

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 9.6, “Attributes for Configuring the JMS Session Pool”, specify the high and low water marks for the endpoint's JMS session pool.

Table 9.6. 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 9.7, “JMS Session Pool Configuration” shows an example of configuring the session pool for a Artix ESB JMS provider endpoint.

Example 9.7. 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>
...
```

Consumer Specific Runtime Configuration

The JMS consumer configuration allows you to specify two runtime behaviors:

- 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 element

You configure consumer runtime behavior using the `jms:clientConfig` element. The `jms:clientConfig` element is a child of the `jms:conduit` element. It has two attributes that are used to specify the configurable runtime properties of a consumer endpoint.

Configuring the response timeout interval

You specify the interval, in milliseconds, a consumer endpoint will wait for a response before timing out using the `jms:clientConfig` element's `clientReceiveTimeout` attribute. The default timeout interval is 2000.

Configure the request time to live

You specify the interval, in milliseconds, that a request can remain unreceived before the JMS broker can delete it using the `jms:clientConfig` element's `messageTimeToLive` attribute. The default time to live interval is 0 which specifies that the request has an infinite time to live.

Example

Example 9.8, "JMS Consumer Endpoint Runtime Configuration" shows a configuration fragment that sets the consumer endpoint's request lifetime to 500 milliseconds and its timeout value to 500 milliseconds.

Example 9.8. JMS Consumer Endpoint Runtime Configuration

```
...
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:clientConfig clientReceiveTimeout="500"
                    messageTimeToLive="500" />
  ...
</jms:conduit>
...
```

Provider Specific Runtime Configuration

The provider specific configuration allows you to specify to runtime behaviors:

- the amount of time a response message can remain unreceived before the JMS broker can delete it.
- the client identifier used when creating and accessing durable subscriptions.

Configuration element

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

Configuring the response time to live

The `jms:serverConfig` element's `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.

Configuring the durable subscriber identifier

The `jms:serverConfig` element's `durableSubscriptionClientId` attribute specifies the client identifier the endpoint uses to create and access durable subscriptions.

Example

Example 9.9, “Provider Endpoint Runtime Configuration” shows a configuration fragment that sets the provider endpoint's response lifetime to 500 milliseconds and its durable subscription client identifier to `jms-test-id`.

Example 9.9. Provider Endpoint Runtime Configuration

```
...
<jms:destination
name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-destination">
  <jms:address ... >
    ...
  </jms:address>
  ...
  <jms:serverConfig messageTimeToLive="500"
                    durableSubscriptionClientId="jms-test-id" />
  ...
</jms:destination>
...
```

Chapter 10. Using WebSphere MQ

Summary

Artix ESB connects to WebSphere MQ using MQ's JMS APIs. It is set up using the standard Artix ESB JMS transport configuration.

Overview

To configure an endpoint to use WebSphere MQ you need to provide the following information:

- The class name of MQ's initial context factory.
- The URL of MQ's JNDI provider.

Important

In addition to the above, you will also need to provide the standard JMS configuration information.

Tip

This information can be provided as part of an endpoint's WSDL document or in an endpoint's configuration.

JMS Addressing Information

Regardless of the JMS provider in use, you will always need to provide some standard addressing information using the `jms:address` element's attributes. Table 10.1, "`jms:address` Attributes for Using WebSphere MQ" shows the attributes needed when using WebSphere MQ's JMS interface.

Table 10.1. `jms:address` Attributes for Using WebSphere MQ

Attribute	Description
`destinationStyle`	WebSphere MQ supports both queues and topics.
`jndiConnectionFactoryName`	The JNDI name for the connection factory can be any string. You will need to use this value when providing the WebSphere MQ specific JMS properties.

Attribute	Description
jndiDestinationName	The JNDI name for the destination can be any string. You will need to use this value when providing the IBM WebSphere MQ specific JMS properties.

The JNDI Initial Context Factory

You specify the WebSphere MQ JNDI initial context factory using a `jms:JMSNamingProperty` element. As shown in Example 10.1, "Specifying the JNDI Initial Context Factory", the value of the `name` attribute is `java.naming.factory.initial` and the value of the `value` attribute is `com.ibm.mq.jms.context.WMQInitialContextFactory`.

Example 10.1. Specifying the JNDI Initial Context Factory

```
<jms:address ...>
  <jms:JMSNamingProperty name="java.naming.factory.initial"
                         value="com.ibm.mq.jms.context.WMQInitialContextFactory" />
  ...
</jms:address>
```

Important

`com.ibm.mq.jms.context.WMQInitialContextFactory` is only available in the IBM supplied SupportPac ME01.

The JNDI Provider URL

You specify the JNDI provider's URL using a `jms:JMSNamingProperty` element. The value of the `name` attribute is `java.naming.provider.url`. The value of the `value` attribute is the URL at which WebSphere MQ's broker is running.

There are two options for a JNDI provider when using WebSphere MQ:

- The default WebSphere MQ installation includes JNDI providers for local filesystems and LDAP servers.
- SupportPac ME01, available from IBM, provides support for using a WebSphere MQ queue manager as a JNDI repository. It can dynamically generate JMS administrable objects, based on actual queues on the queue manager.

For more information about setting up JNDI providers for use with WebSphere MQ, see the WebSphere MQ documentation.

Chapter 11. Using FTP

Summary

Artix allows endpoints to communicate using a remote FTP server as an intermediary persistent datastore. When using the FTP transport, client endpoints will put request messages into a folder on the FTP server and then begin scanning the folder for a response. Server endpoints will scan the request folder on the FTP server for requests. When a request is found, the service endpoint will get it and process the request. When the service endpoint finishes processing the request, it will post the response back to the FTP server. When the client sees the response, it will get the response from the FTP server.

Because of the file-based nature of the FTP transport and the fact that endpoints do not have a direct connection to each other, the FTP transport places the burden of implementing a request/response coordination scheme on the developer. The FTP transport also requires that you implement the logic determining how the request and response messages are cleaned-up.

Adding an FTP Endpoint Using WSDL

Overview

You define an FTP endpoint using WSDL extensions that are placed within a the `port` element of a contract.

The WSDL extensions provided by Artix allow you to specify a number of properties for establishing the FTP connection. In addition, they allow you to specify some of the properties used to define the naming properties for the files used by the transport.

Namespace

To use the FTP transport, you need to describe the endpoint using the FTP WSDL extensions in the physical part of a WSDL document. The extensions used to describe a FTP port are defined in the following namespace:`xmlns:ftp="http://schemas.iona.com/transports/ftp"`

This namespace will need to be included in your contract's `definitions` element.

Defining the connection details

The connection details for the endpoint are defined in an `ftp:port` element. The `ftp:port` element has two attributes that are used to specify the location of the FTP deamon's location: `host` and `port`.

- The `host` attribute is required. It specifies the name of the machine hosting the FTP server to which the endpoint connects.
- The `port` attribute is optional. It specifies the port number on which the FTP server is listening. The default value is `21`.

Example 11.1, "Defining an FTP Endpoint" shows an example of a `port` element defining an FTP endpoint.

Example 11.1. Defining an FTP Endpoint

```
<port name="FTPendpoint">
  <ftp:port host="Dauphin" port="8080" />
</port>
```

In addition to the two required attributes, the `ftp:port` element has the following optional attributes:

Table 11.1. Optional Attributes for `ftp:port`

Attribute	Description
`requestLocation`	Specifies the location on the FTP server where requests are stored. The default is `/`.
`replyLocation`	Specifies the location on the FTP server where replies are stored. The default is `/`.
`connectMode`	Specifies the connection mode used to connect to the FTP daemon. Valid values are `passive` and `active`. The default is `passive`.
`scanInterval`	Specifies the interval, in seconds, at which the request and reply locations are scanned for updates. The default is `5`.

Specifying optional naming properties

You can specify optional naming policies using an `ftp:properties` element. The `ftp:properties` element is a container for a number of `ftp:property` elements. The `ftp:property` elements specify the individual naming properties. Each `ftp:property` element has two attributes, `name` and `value`, that make up a name-value pair that are used to provide information to the naming implementation used by the endpoint.

The default naming implementation provided with Artix has two properties:

Property	Description
staticFilenames	Determines if the endpoint uses a static, non-unique, naming scheme for its files. Valid values are `true` and `false`. The default is `true`.
requestFilenamePrefix	Specifies the prefix to use for file names when `staticFilenames` is set to `false`.

For information on defining optional properties see Using Properties to Control Coordination Behavior .

Adding an Configuration for an FTP Endpoint

Overview

There are a number of configurable properties that do not make sense to set in an application's WSDL document. These include the username and password used to login to the FTP server and some of the connection's timeout settings. These properties, along with the message coordination logic, are able to be set in the applications configuration file.

Namespaces

The configuration elements used to configure an FTP endpoint are defined in the namespace `http://schemas.iona.com/soa/ftp-config`.

This namespace will need to be added to the list of namespaces in the `schemaLocation` attribute of your configuration's `bean` element. In addition, you should also add a namespace shortcut for the namespace. Example 11.2, “Namespace Declarations for FTP Configuration” shows a configuration `bean` element with the proper attributes.

Example 11.2. Namespace Declarations for FTP Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ftp-conf="http://schemas.iona.com/soa/ftp-config"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd

                             http://schemas.iona.com/soa/ftp-config
                             http://schemas.iona.com/soa/ftp-config.xsd">
```

Consumer configuration

Consumer endpoints are configured using the `ftp-conf:conduit` element. Using this element you can configure the following FTP endpoint properties:

- FTP connection properties
- the credentials used to access the FTP server
- the classes used for the consumer's message coordination logic

Example 11.3, “FTP Consumer Configuration” shows the configuration for a consumer endpoint. The consumer's FTP endpoint is configured to use an active connection and scan for new files every three seconds.

Example 11.3. FTP Consumer Configuration

```
<beans ...
      xsi:schemaLocation="http://schemas.iona.com/soa/ftp-config
         http://schemas.iona.com/soa/ftp-config.xsd
         http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

<ftp-conf:conduit id="{http://iona.com/soap_over_ftp}FTPPort.ftp-conduit">
  <ftp-conf:connection connectMode="active"
                       scanInterval="3" />
</ftp-conf:conduit>
```

Provider configuration

Provider endpoints are configured using the `ftp-conf:destination` element. Using this element you can configure the following FTP endpoint properties:

- FTP connection properties
- the credentials used to access the FTP server
- the classes used for the provider's message coordination logic

Example 11.4, “FTP Provider Configuration” shows the configuration for a provider endpoint. The provider's FTP endpoint is configured to timeout if a connection cannot be established after 5 seconds and connect to the FTP server using the username "JoeFred".

Example 11.4. FTP Provider Configuration

```
<beans ...
       xsi:schemaLocation="http://schemas.iona.com/soa/ftp-config
         http://schemas.iona.com/soa/ftp-config.xsd
         http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

<ftp-conf:destination id="{http://iona.com/soap_over_ftp}FTPPort.ftp-destination">
  <ftp-conf:connection connectTimeout="5000" />
  <ftp-conf:credentials name="JoeFred"
                        password="FredJoe" />
</ftp-conf:conduit>
```

Connection configuration

The FTP transport connection information is configurable by adding a `ftp-conf:connection` child element to an endpoint's `ftp-conf:conduit` element or `ftp-conf:destination` element. The `ftp-conf:connection` element's attributes, described in Table 11.2, “Attributes for ftp-conf:connection”, are used to specify the connection setting information.

Table 11.2. Attributes for `ftp-conf:connection`

Attribute	Description
`connectMode`	Specifies if the endpoint connects to the FTP server using an active or a passive connection. Valid values are `passive`(default) or `active`.
`connectTimeout`	Specifies a timeout value in milliseconds for establishing a connection with a remote FTP daemon. The default is `-1` which specifies that endpoint will never timeout.
`recieveTimeout`	Specifies a receive timeout value in milliseconds for the FTP daemon filesystem scanner. The receive timeout will occur when the following condition is met: `CurrentTime - StartReplyScanningTime >= plugins:ftp:policy:connection:receiveTimeout` It is recommended that the receive timeout value is greater than `scanInterval` * 1000. If this value is set to `0`, it is guaranteed that there will be at least one scan of the remote FTPD filesystem before the timeout. The default is `-1` which specifies that the endpoint will never timeout.

Attribute	Description
scanInterval	Specifies the interval, in seconds, at which the request and reply locations are scanned for updates. The default is five seconds.
useFilenameMaskOnScan	Specifies whether the Artix ESB Java Runtime uses a filename mask when calling the FTP daemon with a FTP **LIST** command (for example, **LIST myrequests***). Some FTP daemons do not implement support for listing a subset of files based on a filename mask. To enable interoperability with such servers, this variable must be set to false. However, if you know that an FTP daemon supports a filtered **LIST** command, setting this variable to true increases FTP transport performance. The default is false.

Example 11.5, "Configuring the FTP Connection Properties" shows the configuration for a consumer endpoint that uses the filename mask optimization and has a receive timeout of ten seconds.

Example 11.5. Configuring the FTP Connection Properties

```
<beans ...
      xsi:schemaLocation="http://schemas.iona.com/soa/ftp-config
         http://schemas.iona.com/soa/ftp-config.xsd
         http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<ftp-conf:conduit id="{http://iona.com/soap_over_ftp}FTPPort.ftp-conduit">
  <ftp-conf:connection recieveTimeout="10000"
                       useFilenameMaskOnScan="true" />
</ftp-conf:conduit>
```

Login configuration

The FTP transport default behavior is to login to the FTP server as anonymous. You can specify a username and password for accessing the FTP server using the ftp-conf:credentials element. The ftp-conf:credentials element is a child of both the ftp-conf:conduit element and the ftp-conf:destination element. It has two attributes, described in Table 11.3, "Attributes for ftp-conf:credentials", that specify the username and password.

Table 11.3. Attributes for `ftp-conf:credentials`

Attribute	Description
`name`	Specifies the username used to login to the FTP server.  **Important** This user must have the required credentials to list, add, move and remove files from the filesystem location specified by the application's WSDL document.
`password`	Specifies the password used to login to the FTP server.

Coordinating Requests and Responses

Introduction

Overview

FTP requires that messages are written out to a file system for retrieval. This poses a few problems. The first is determining a naming scheme that is agreed upon by all endpoints that use a common location on an FTP server. Client endpoints and the server endpoints they are making requests on need a method to coordinate requests and responses. This includes knowing which messages are intended for which endpoint.

The other problem posed by using a file system as a transport is knowing when a message can be cleaned-up. If a message is cleaned-up too soon, there is no way to re-read the message if something goes wrong while it is being processed. If a message is not cleaned-up soon enough, it is possible that the message will be processed more than once.

Artix requires that you implement the logic used to determine the file naming and clean-up logic used by your FTP endpoints. This is done by implementing four Java interfaces: two for the client-side and two for the server-side.

Default implementation

Artix provides a default implementation for coordinating requests and responses. The default implementation enables clients and servers to interact as if they are using a standard RPC mechanism. Message names are generated at runtime following a pattern based on the server endpoint's service name. Request messages are cleaned-up by the server endpoint when the corresponding response is written to the file system. Responses are cleaned-up by the client endpoint when they are read from the file system.

Implementing the Consumer's Coordination Logic

Overview

The consumer-side of the coordination implementation is made up of two parts:

- The filename factory is responsible for generating the filenames used for storing request messages on the FTP server and determining the name of the associated replies.
- The reply lifecycle policy is responsible for cleaning-up reply files.

The filename factory

The consumer-side filename factory is created by implementing the interface `com.iona.cxf.transport.ftp.filenamepolicy.client.FilenameFactory`. Example 11.6, "Client-Side Filename Factory Interface" shows the interface.

Example 11.6. Client-Side Filename Factory Interface

```
package com.iona.cxf.transport.ftp.filenamepolicy.client;

import java.util.Properties;

import javax.xml.namespace.QName;

import com.iona.cxf.transport.ftp.filenamepolicy.FilenameFactoryPropertyMetaData;

public interface FilenameFactory
{

   void  initialize(QName service, String port, Properties properties) throws Exception;


    String getNextRequestFilename() throws Exception;

    String getRequestIncompleteFilename(String requestFilename) throws Exception;

    String getReplyFilename(String requestFilename) throws Exception;

    FilenameFactoryPropertyMetaData[] getPropertiesMetaData();
};
```

The interface has four methods to implement:

`initialize()`

`initialize()` is called by the transport when it is loaded. It receives the following:

- the QName of the service the client on which the consumer wants to make requests.
- the value of the `name` attribute for the `wsdl:port` element defining the endpoint implementing the service.
- an array containing any properties you specified as `ftp:property` elements in your client's contract.

This method is used to set up any resources you need to implement naming scheme used by the consumer-side endpoints. For example, the default implementation uses `initialize()` to do the following:

1. Determine if the user wants to use static filenames based on an `ftp:property` element in the contract. For more information see Using Properties to Control Coordination Behavior.
2. If so, it generates a static filename prefix for the requests.
3. If not, it uses the user supplied filename prefix for the requests.

`getNextRequestFilename()`

`getNextRequestFilename()` is called by the transport each time a request is sent out. It returns a string that the transport will use as the filename for the completed request message. For example, the default implementation creates a filename by appending a string representing the server endpoint's system address and the system time, in hexcode, to the prefix generated in `initialize()`.

`getRequestIncompleteFilename()`

`getRequestIncompleteFilename()` is called by the transport each time a request is sent out. It returns a string that the transport will use as the filename for the request message as it is being transmitted. For example, the default implementation creates a filename by appending a the request filename with `_incomplete`.

`getReplyFilename()`

`getReplyFilename()` is called by the transport when it starts listening for a response to a two-way request. It receives a string representing the name of the request's filename. It returns the name of the file that will contain the response to the specified request. For

example, the default implementation generates the reply filename by appending `_reply` to the request filename.

The reply lifecycle policy

The reply lifecycle policy is created by implementing the `com.iona.cxf.transport.ftp.filenamepolicy.FileLifecycle` interface. Example 11.7, "Reply Lifecycle Interface" shows the interface.

Example 11.7. Reply Lifecycle Interface

```
package com.iona.cxf.transport.ftp.filenamepolicy;

public interface FileLifecycle
{
  boolean shouldDeleteFile(String fileName) throws Exception;

  String renameFile(String fileName) throws Exception;
}
```

The interface has two methods to implement:

`shouldDeleteFile()` — `shouldDeleteFile()` is called by the transport after it completes reading in a reply. It receives the filename of the reply and returns a boolean stating if the file should be deleted. If `shouldDeleteFile()` returns true, the transport deletes the reply file. If it returns false, the transport renames reply file based on the logic implemented in `renameFile()`.

`renameFile()` — `renameFile()` is called by the transport if `shouldDeleteReplyFile()` returns false. It receives the original name of the reply file. It returns a string the contains the filename the transport uses to rename the reply file.

Configuring the client's coordination logic

If you choose to implement your own coordination logic for an FTP client endpoint, you need to configure the endpoint to load the your implementation classes. This is done by adding the `ftp-conf:clientNaming` element to the endpoint's configuration. The `ftp-conf:clientNaming` element's attributes are described in Table 11.4, "Attributes for the ftp-conf:clientNaming Element".

Table 11.4. Attributes for the `ftp-conf:clientNaming` Element

Attribute	Description
`filenameFactory`	Specifies the name of the class implementing the client's filename factory.
`replyFileLifecycle`	Specifies the name of the class implementing the client's reply lifecycle policy.

Important

Both classes need to be on the endpoint's classpath.

Example 11.8, "Configuring an FTP Client Endpoint's Naming Policy" shows an example of a configuration fragment that specifies an FTP client endpoint's coordination policies.

Example 11.8. Configuring an FTP Client Endpoint's Naming Policy

```
<beans ...
       xsi:schemaLocation="http://schemas.iona.com/soa/ftp-config
         http://schemas.iona.com/soa/ftp-config.xsd
         http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<ftp-conf:conduit id="{http://iona.com/soap_over_ftp}FTPPort.ftp-conduit">
  <ftp-conf:clientNaming filenameFactory="demo.ftp.policy.client.myFilenameFactory"
                      replyFileLifecycle="demo.ftp.policy.client.myReplyFileLifecycle"
 />
</ftp-conf:conduit>
```

For more information on configuring Artix ESB Java Runtime see Configuring and Deploying Artix Solutions, Java Runtime [../../deploy/java/index.htm].

Implementing the Server's Coordination Logic

Overview

The server-side of the coordination implementation is made up of two parts:

- The filename factory is responsible for identifying which requests to dispatch and how to name reply messages.
- The request lifecycle policy is responsible for cleaning-up request files.

The filename factory

The server-side filename factory is created by implementing the interface `com.iona.cxf.transport.ftp.filenamepolicy.server.FilenameFactory`. Example 11.9, "Server-Side Filename Factory Interface" shows the interface.

Example 11.9. Server-Side Filename Factory Interface

```
package com.iona.cxf.transport.ftp.filenamepolicy.server;

import java.util.Properties;

import javax.xml.namespace.QName;
import com.iona.cxf.transport.ftp.filenamepolicy.FilenameFactoryPropertyMetaData;
import com.iona.cxf.transport.ftp.ftpclient.Element;

public interface FilenameFactory
{
  void initialize(QName service, String port, Properties properties) throws Exception;


  String getRequestFilenamesRegEx() throws Exception;

  Element[] updateRequestFiles(Element[] inElements) throws Exception;

  String getReplyFilename(String requestFilename) throws Exception;

  FilenameFactoryPropertyMetaData[] getPropertiesMetaData();
}
```

The interface has six methods to implement:

`initialize()`

`initialize()` is called by the transport when it is activated. It receives the following:

- the QName of the service to which the endpoint is implementing.
- the value of the `name` attribute for the `port` element defining the endpoint's connection details.
- an array containing any properties you specified as `ftp:property` elements in your server endpoint's contract.

	This method is used to set up any resources you need to implement naming scheme used by the server-side endpoints. For example, the default implementation uses `initialize()` to do the following: 1. Determine if the user wants to use static filenames based on an `ftp:property` element in the contract. For more information see Using Properties to Control Coordination Behavior. 2. If so, it generates a static filename prefix for the requests. 3. If not, it uses the user supplied filename prefix for the requests.
`getRequestFilenamesRegEx()`	`getRequestFilenamesRegEx()` is called by the transport when it initializes the server-side FTP listener. It returns a regular expression that is used to match request filenames intended for a specific server instance. For example, the default implementation returns a regular expression of the form `{wsdl:tns}_{wsdl:service(@name)}_{wsdl:port(@name)}_{reqUuid}`.
`updateRequestFiles()`	`updateRequestFiles()` is called by the transport after it determines the list of possible requests and before it dispatches the requests to the service implementation for processing. It receives an array of `com.iona.cxf.transport.ftp.ftpclient.Element` objects. This array is a list of all the request messages selected by the request filename regular expression. `updateRequestFiles()` returns an array of `Element` objects containing only the messages that are to be dispatched to the service implementation.
`getReplyIncompleteFilename()`	`getReplyInclompleteFilename()` is called by the transport when it is ready to post a response. It receives the filename of the request that generated the response. It returns a string that is used as the filename for the response as it is being written to the FTP server. For example, the default implementation returns `_incomplete` appended to request filename.
`getReplyFilename()`	`getReplyFilename()` is called by the transport after it finishes writing a response to the FTP server. It receives the filename of the request that generated the response. It returns a string that is used as the filename for the completed response. For example, the default implementation returns `_reply` appended to request filename.

`getPropertiesMetaData()` `getPropertiesMetaData()` is a convenience function that returns an array of all the possible properties you can use to effect the behavior of the FTP naming scheme. The properties returned correspond to the values defined in the `ftp:properties` element. For more information see Using Properties to Control Coordination Behavior.

The request lifecycle policy

The request lifecycle policy is created by implementing the `com.iona.cxf.transport.ftp.filenamepolicy.FileLifecycle` interface. Example 11.10, "Request Lifecycle Interface" shows the interface.

Example 11.10. Request Lifecycle Interface

```
package com.iona.cxf.transport.ftp.filenamepolicy;

public interface FileLifecycle
{
  boolean shouldDeleteFile(String fileName) throws Exception;

  String renameFile(String fileName) throws Exception;
}
```

The interface has two methods to implement:

`shouldDeleteFile()` `shouldDeleteFile()` is called by the transport after it completes writing in a response. It receives the filename of the request that generated the response and returns a boolean stating if the file should be deleted. If `shouldDeleteFile()` returns true, the transport deletes the request file. If it returns false, the transport renames reply file based on the logic implemented in `renameFile()`.

`renameFile()` `renameFile()` is called by the transport if `shouldDeleteFile()` returns false. It receives the original name of the request file. It returns a string the contains the filename the transport uses to rename the request file.

Configuring the server's coordination logic

If you choose to use your own coordination logic for an FTP server endpoint, you need to configure the endpoint to load the proper implementation classes. This is done by adding a `ftp-conf:serverNaming`

element the endpoint's destination configuration. The `ftp-conf:serverNaming` element's attributes are described in Table 11.5, "Attributes of the ftp-conf:serverNaming Element".

Table 11.5. Attributes of the `ftp-conf:serverNaming` Element

Attribute	Description
`filenameFactory`	Specifies the name of the class implementing the server's filename factory.
`requestFileLifecycle`	Specifies the name of the class implementing the server's request lifecycle policy.

Important

Both classes need to be on the endpoint's classpath.

Example 11.11, "Configuring an FTP Server Endpoint's Naming Policy" shows an example of a configuration fragment that specifies an FTP server endpoint's coordination policies.

Example 11.11. Configuring an FTP Server Endpoint's Naming Policy

```
<beans ...
       xsi:schemaLocation="http://schemas.iona.com/soa/ftp-config
         http://schemas.iona.com/soa/ftp-config.xsd
         http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

<ftp-conf:destination id="{http://iona.com/soap_over_ftp}FTPPort.ftp-destination">
  <ftp-conf:serverNaming filenameFactory="demo.ftp.policy.server.myFilenameFactory"

requestFileLifecycle="demo.ftp.policy.server.myReplyFileLifecycle" />
</ftp-conf:destination>
```

For more information on configuring Artix ESB Java Runtime see Configuring and Deploying Artix, Java Runtime [../../deploy/java/index.htm].

Using Properties to Control Coordination Behavior

Overview

In order to ensure that your FTP client endpoints and FTP server endpoints are using the same coordination behavior, you may need to pass some information to the transports as they initialize. To make this information available to both sides of the application and still be settable at run time, the FTP transport allows you to

provide custom properties that are settable in an endpoint's contract. These properties are set using the `ftp:properties` element.

Properties in the contract

You can place any number of custom properties into port element defining an FTP endpoint. As described in Specifying optional naming properties, the `ftp:properties` element is a container for one or more `ftp:property` elements. The `ftp:property` element has two attributes: `name` and `value`. Both attributes can have any string as a value. Together they form a name/value pair that your coordination logic is responsible for processing.

For example, imagine an FTP endpoint defined by the `port` element in Example 11.12, "FTP Endpoint with Custom Properties".

Example 11.12. FTP Endpoint with Custom Properties

```
<port ...>
  <ftp:port ... />
  <ftp:properties>
    <ftp:property name="UseHumanNames" value="true" />
    <ftp:property name="LastName" value="Doe" />
  </ftp:properties>
</port>
```

The endpoint is configured using two custom FTP properties:

- `UseHumanNames` with a value of `true`.
- `LastName` with a value of `Doe`.

These properties are only meaningful if the coordination logic used by the endpoint supports them. If they are not supported, they are ignored.

Supporting the properties

The `initialize()` method of both the client-side filename factory and the server-side filename factory take a `java.util.Properties` object. The `Properties` object is populated by the contents of the endpoints `ftp:properties` element when the transport is initialized.

The `Properties` object can be used to access all of the properties defined by `ftp:property` elements. To access the properties you can use either of the `getProperty()` methods to extract the value. Once you have the values of the properties, it is up to you to determine how they impact the coordination scheme.

Example 11.13, “Using Custom FTP Properties” shows code for supporting the properties shown in Example 11.12, “FTP Endpoint with Custom Properties”.

Example 11.13. Using Custom FTP Properties

```
import java.util.Properties;

String nameTypeProp = "UseHumanNames";
String lastNameProp = "LastName";

  String useNames = (string)properties.getProperty(nameTypeProp);

  if ("TRUE".equalsIngnoreCase(useNames))
  {
    boolean useHumanNames = true;
    String lastName = properties.getProperty(lastNameProp);  }
  }
  else
  {
    boolean useHumanNames = false;
  }
}
```

Filling in the filename factory property metadata

The server-side filename factory's `getPropertiesMetaData()` method is a convenience function that can be used to publish the supported custom properties. It returns the details of the supported properties in an array of `com.iona.cxf.transport.ftp.filenamepolicy.FilenameFactoryPropertyMetaData` objects.

`FilenameFactoryPropertyMetaData` objects have three fields:

- `name` is a string that specifies the value of the `ftp:property` element's `name` attribute.
- `readOnly` is a boolean that specifies if you can set this property in a contract.
- `valueSet` is an array of strings that specify the possible values for the property.

`FilenameFactoryPropertyMetaData` objects do not have any methods for populating its fields once the object is instantiated. You must set all of the values using the constructor that is shown in Example 11.14, "Constructor for `FilenameFactoryPropertyMetaData`" .

Example 11.14. Constructor for `FilenameFactoryPropertyMetaData`

```
public FilenameFactoryPropertyMetaData(String n, boolean ro,
                                       String[] vs)
{
  name = n;
  readOnly = ro;
  valueSet = vs;
}
```

Example 11.15, "Populating the Filename Properties Metadata" shows code for creating an array to be returned from `getPropertiesMetaData()`.

Example 11.15. Populating the Filename Properties Metadata

```
FilenameFactoryPropertyMetaData[] propMetas = new FilenameFactoryPropertyMetaData[]
{
  new FilenameFactoryPropertyMetaData("UseHumanNames", false,
                                      new String[] {Boolean.TRUE.toString(),
                                                    Boolean.FALSE.toString()}),
  new FilenameFactoryPropertyMetaData("LastName", false, null)
};
```

The list of possible values specified for the property LastName is set to `null` because the property can have any string value.

Index

J

M

N

P

S

W

X