Artix 5.6.3

Bindings and Transports, Java Runtime

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2015-03-11

Contents

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 describes how to work with the message bindings.
- Part II 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 Library***.

Further Information and Product Support

Additional technical information or advice is available from several sources.

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

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

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

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

Information We Need

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

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

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

Additional technical information or advice is available from several sources.

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

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

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx (documentation updates and PDFs)

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

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

Part I

Bindings

In this part

This part contains the following chapters:

Understanding Bindings in WSDL

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

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 `WSDLbinding` 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 in WSDL.

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 command line tools for adding bindings to your contracts.

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

Using SOAP 1.1 Messages

Artix 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. In addition, you can define a SOAP binding that uses MIME multipart attachments.

Adding a SOAP 1.1 Binding

Artix provides the **wsdltosoap** tool to add a SOAP 1.1 binding for a logical interface.

Using wsdltosoap

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

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

The command has the following options:

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

The command has the following optional arguments:

Option	Description
`-b` *`binding-name`*	Specifies the name of the generated SOAP binding.
`-d` *`output-directory`*	Specifies the directory to place the generated WSDL.
`-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	Description
`-style (document/ rpc)`	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is to `document`.
`-use (literal/ encoded)`	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is `literal`.
`-v`	Displays the version number for the tool.
`-verbose`	Displays comments during the code generation process.
`-quiet`	Suppresses comments during the code generation process.

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

Important: wsdltosoap does not support the generation of `document/encoded` SOAP bindings.

For more information see *wsdl2soap* in the ***Artix ESB Java Runtime Command Reference***.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it would be defined in a WSDL fragment similar to the o ne shown in Example 1.

Example 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.

Example 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

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 3. 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 i nto 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 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 providesa 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.

Example

Example 4 shows a modified version of the `orderWidgets` service shown in Example 1. This version has been modified so that each order has an xsd:base64binary value placed in the SOAP header of the request and response. The SOAP header is defined as being the `keyVal` part from the `widgetKey` message. In this case you are responsible for adding the SOAP header to your application logic because it is not part of the input or output message.

Example 4. SOAP 1.1 Binding with a SOAP Header

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

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

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

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

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

```
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

You can modify Example 4 so that the header value is a part of the input and output messages as shown in Example 5. In this case `keyVal` is a part of the input and output messages. In the `soap:body` element's `parts` attribute specifies that `keyVal` cannot be inserted into the body. However, it is inserted into the SOAP header.

Example 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>
```

Using SOAP 1.2 Messages

Artix 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

Artix provides the **wsdltosoap** tool to add a SOAP 1.2 binding for a logical interface.

Using wsdltosoap

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

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

The tool has the following required arguments:

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

The tool has the following optional arguments:

Option	Interpretation
`-b` *`binding-name`*	Specifies the name of the generated SOAP binding.
`-soap12`	Specifies that the generated binding will use SOAP 1.2.
`-d` *`output-directory`*	Specifies the directory to place the generated WSDL file.

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

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

Important: wsdltosoap does not support the generation of document/encoded SOAP bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a WSDL fragment similar to the one shown in Example 6.

Example 6. 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 7 on page 46.

Example 7. 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

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

Syntax

The syntax for defining a SOAP header is shown in Example 8.

Example 8. 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 1.

Table 1. soap12:header Attributes

Attribute	Description
message	A required attribute specifying the qualified name of the message from which the part being inserted into the header is taken.
part	A required attribute specifying the name of the message part inserted into the SOAP header.
use	Specifies if the message parts are to be encoded using encoding rules. If set to encoded the message parts are encoded using the encoding rules specified by the value of the encodingStyle attribute. If set to literal, the message parts are defined by the schema types referenced.
encodingStyle	Specifies the encoding rules used to construct the message.
namespace	Defines the namespace to be assigned to the header element serialized with use="encoded".

Splitting messages between body and header

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

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

Note: When you define a SOAP header using parts of the parent message, Artix ESB automatically fills in the SOAP headers for you.

Example

Example 9 shows a modified version of the orderWidgets service shown in Example 6. This version is modified so that each order has an xsd:base64binary value placed in the header of the request and the response. The header is defined as being the keyVal part from the widgetKey message. In this case you are responsible for adding the application logic to create the header because it is not part of the input or output message.

Example 9. SOAP 1.2 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
   targetNamespace="http://widgetVendor.com/widgetOrderForm"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
   xmlns:tns="http://widgetVendor.com/widgetOrderForm"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
   xmlns:SOAP-
   ENC="http://schemas.xmlsoap.org/soap/encoding/">
<types>
  <schema
  targetNamespace="http://widgetVendor.com/types/widgetTypes"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
```

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

You can modify Example 9 so that the header value is a part of the input and output messages, as shown in Example 10. In this case `keyVal` is a part of the input and output messages. In the `soap12:body` elements the `parts` attribute specifies that `keyVal` should not be inserted into the body. However, it is inserted into the header.

Example 10. 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/widgetTy
          pes" 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>
```

Sending Binary Data Using SOAP Attachments

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

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.

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 11.

Example 11. 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 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 2. `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.

Attribute	Description
`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 and Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.

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

Example

Example 12 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 p art, `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 12. 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>
```

Sending Binary Data with SOAP MTOM

SOAP Message Transmission Optimization Mechanism (MTOM) is a mechanism for transmitting binary data in SOAP messages. 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 a method for sending binary data. MTOM uses 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 `DataHandlers` is beyond the scope of this book.

Annotating Data Types to use MTOM

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

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

WSDL first

Example 13 shows a WSDL document for a Web service that uses a message which contains one string field, one integer field, and a binary field. The binary field is intended to carry a large image file, so it is not appropriate to send it as part of a normal SOAP message.

Example 13. Message for MTOM

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

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

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

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

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

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

Note: The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and are described in detail in Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies and Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.

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

Example 14 shows how you can modify xRayType from Example 13 for using MTOM.

Example 14. Binary Data for MTOM

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

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

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

Java first

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

Make sure the field holding the binary data is a DataHandler.

Add the @XmlMimeType() annotation to the field containing the data you want to stream as an MTOM attachment.

Example 15 shows a JAXB class annotated for using MTOM.

Example 15. 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 sends all binary data as either part of the normal SOAP message or as an unoptimized attachment. You can activate MTOM support either programmatically or through the use of configuration.

Using JAX-WS APIs

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

Service provider

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

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

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

2. Get the SOAP binding from the `Endpoint` using its `getBinding()` method, as shown in Example 16.

Example 16. Getting the SOAP Binding from an Endpoint

```
// Endpoint ep is declared previously
```

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

Set the binding's MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method, as shown in Example 17 on page 51.

Example 17. Setting a Service Provider's MTOM Enabled Property

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

Consumer

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

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

TIP: For information on getting a consumer proxy see *Developing a Consumer Without a WSDL Contract* in ***Developing Artix® Applications with JAX-WS*** or *Developing a Consumer 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 18.

Example 18. Getting a SOAP Binding from a BindingProvider

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

Set the bindings MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method, as shown in Example 19 on page 52.

Example 19. Setting a Consumer's MTOM Enabled Property

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

Using configuration

If you publish your service using XML, such as when deploying to a container, you can enable your endpoint's MTOM support in the endpoint's configuration file. For more information on configuring endpoints see the ***Artix ESB Deployment Guide***.

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.

Add a `entry` child element to the jaxws:property element.

Set the entry element's key attribute to mtom-enabled.

Set the entry element's `value` attribute to `true`.

Example

Example 20 shows an endpoint that is MTOM enabled.

Example 20. 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>
```

Using XML Documents

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

XML binding namespace

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

Hand editing

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

1. Add the namespace declaration to include the Artix ESB 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 ESB.

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 23 would generate an XML document with the root element `lineNumber`.

Example 21. 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 ESB will always generate a valid XML document even if the `rootNode` attribute is not set. However, the message in Example 24 would generate an invalid XML document.

Example 22. 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 25 for the message defined in Example 24. The Artix-generated XML document is invalid because it has two root elements: `pairName` and `entryNum`.

Example 23. Invalid XML Document

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

If you set the `rootNode` attribute, as shown in Example 26 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 24. 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 27. Notice that the XML document now only has one root element.

Example 25. 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 w anted the output message defined in Example 26 to have a different root element from the input message, you could override the binding's root element as shown in Example 28.

Example 26. 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

In this part

This part contains the following chapters:

How Endpoints are Defined in WSDL

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

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 command line tools for adding a number of the endpoint types to your contracts.

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

Artix ESB endpoint definitions are built using extensions defined for each of the transports Artix ESB Java Runtime supports. This includes the following transports:

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

Using HTTP

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

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 `soap: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 29 shows a `port` element used to send SOAP 1.1 messages over HTTP.

Example 27. 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 `wsoap12: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 30 shows a `port` element used to send SOAP 1.2 messages over HTTP.

Example 28. 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>
```

Configuring a Consumer

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

A consumer endpoint can be configured using two mechanisms:

- Configuration
- WSDL

Using Configuration

Namespace

The elements used to configure an HTTP consumer endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the lines shown in Example 29 to the beans element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 29. 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, `name`, that specifies the WSDL `port` element corresponding to the endpoint. The value for the name attribute takes the form `portQName.http-conduit`. Example 30 shows the `http-conf:conduit` element that would be used to add configuration for an endpoint that is specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> when the endpoint's target namespace is `http://widgets.widgetvendor.net`.

Example 30. http-conf:conduit Element

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

The `http-conf:conduit` element has child elements that specify configuration information. They are described in Table 3.

Table 3. 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 parameters for configuring the basic authentication method that the endpoint uses preemptively. The preferred approach is to supply a Basic Authentication Supplier object.
`http-conf:proxyAuthorization`	Specifies the parameters for configuring basic authentication against outgoing HTTP proxy servers.
`http-conf:tlsClientParameters`	Specifies the parameters used to configure SSL/TLS.
`http-conf:basicAuthSupplier`	Specifies the bean reference or class name of the object that supplies the basic authentication information used by the endpoint, either preemptively or in response to a `401` HTTP challenge.

Element	Description
`http-conf:trustDecider`	Specifies the bean reference or class name of the object that checks the HTTP(S) `URLConnection` object to establish trust for a connection with an HTTPS service provider before any information is transmitted.

The client element

The `http-conf:client` element is used to configure the non-security properties of a consumer endpoint's HTTP connection. Its attributes, described in Table 4, specify the connection's properties

Table 4 HTTP Consumer Configuration Attributes

Attribute	Description
`ConnectionTimeout`	Specifies the amount of time, in milliseconds, that the consumer attempts to establish a connection before it times out. The default is `30000`. `0` specifies that the consumer will continue to send the request indefinitely.
`ReceiveTimeout`	Specifies the amount of time, in milliseconds, that the consumer will wait for a response before it times out. The default is `30000`. `0` specifies that the consumer will wait indefinitely.
`AutoRedirect`	Specifies if the consumer will automatically follow a server issued redirection. The default is `false`.
`MaxRetransmits`	Specifies the maximum number of times a consumer will retransmit a request to satisfy a redirect. The default is `-1` which specifies that unlimited retransmissions are allowed.
`AllowChunking`	Specifies whether the consumer will send requests using chunking. The default is `true`, which specifies that the consumer will use chunking when sending requests. Chunking cannot be used if either of the following are true: • `http-conf:basicAuthSupplier` is configured to provide credentials preemptively. • `AutoRedirect` is set to `true`. In both cases the value of `AllowChunking` is ignored and chunking is disallowed.

Attribute	Description
`Accept`	Specifies what media types the consumer is prepared to handle. The value is used as the value of the HTTP Accept property. The value of the attribute is specified using multipurpose internet mail extensions (MIME) types.
`AcceptLanguage`	Specifies what language (for example, American English) the consumer prefers for the purpose of receiving a response. The value is used as the value of the HTTP `AcceptLanguage` property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, `en-US` represents American English.
`AcceptEncoding`	Specifies what content encodings the consumer is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP AcceptEncoding property.
`ContentType`	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType property. The default is `text/xml`. For web services, this should be set to `text/xml`. If the client is sending HTML form data to a CGI script, this should be set to `application/x-www-form-urlencoded`. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to `application/octet-stream`.
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP Host property. This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).

Attribute	**Description**
`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • `Keep-Alive` — Specifies that the consumer wants the connection kept open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • `close`(default) — Specifies that the connection to the server is closed after each request/response sequence.
`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 on the client that is sending the request.
`Referer`	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property. This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. If the AutoRedirect attribute is set to true and the request is redirected, any value specified in the Referer attribute is overridden. The value of the HTTP Referer property is set to the URL of the service that redirected the consumer’s original request.

Attribute	**Description**
`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. 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 31 shows the configuration of an HTTP consumer endpoint that wants to keep its connection to the provider open between requests, that will only retransmit requests once per invocation, and that cannot use chunking streams.

Example 31. 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/configur
       ation
                  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 referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the line shown in Example 33 to the `definitions` element of your endpoint's WSDL document.

Example 32. HTTP Consumer WSDL Element's Namespace

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

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 4.

Example

Example 34 shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 33. WSDL to Configure an HTTP Consumer Endpoint

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

Consumer Cache Control Directives

Table 5 lists the cache control directives supported by an HTTP consumer.

Table 5. http-conf:client Cache Control Directives

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

Configuring a Service Provider

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

A service provider endpoint can be configured using two mechanisms:

- Configuration
- WSDL

Using Configuration

Namespace

The elements used to configure an HTTP provider endpoint are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix http-conf. In order to use the HTTP configuration elements you must add the lines shown in Example 34 to the `beans` element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 34. HTTP Provider Configuration Namespace

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

The destination element

You configure an HTTP service provider endpoint using the `http-conf:destination` element and its children. The `http-conf:destination` element takes a single attribute, `name`, that specifies the WSDL `port` element that corresponds to the endpoint. The value for the `name` attribute takes the form `portQName.http-destination`. Example 35 shows the `http-conf:destination` element that is used to add configuration for an endpoint that is specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> when the endpoint's target namespace is `http://widgets.widgetvendor.net`.

Example 35. http-conf:destination Element

```
...
  <http-conf:destination
name="{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 6.

Table 6. 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 7, specify the connection's properties.

Table 7. HTTP Service Provider Configuration Attributes

Attribute	Description
`ReceiveTimeout`	Sets the length of time, in milliseconds, the service provider attempts to receive a request before the connection times out. The default is `30000`. `0` specifies that the provider will not timeout.

Attribute	**Description**
`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.

Attribute	Description
`ContentEncoding`	Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`. This value is used as the value of the HTTP ContentEncoding property. The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Artix ESB 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 37 shows the configuration for an HTTP service provider endpoint that honors keep-alive requests and suppresses all communication errors.

Example 36. HTTP Service Provider Endpoint Configuration

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

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

Using WSDL

Namespace

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

Example 37. 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 7.

Example

Example 39 shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 38. 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 lists the cache control directives supported by an HTTP service provider.

Table 8. 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

Directive	Behavior
	response. If no response header fields are specified, the restriction applies to the entire response.
`public`	Any cache can store the response.
`private`	Public (shared) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`no-store`	Caches must not store any part of the response or any part of the request that invoked it.
`no-transform`	Caches must not modify the media type or location of the content in a response between a server and a client.
`must-revalidate`	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.
`proxy-revalidate`	Does the same as `must-revalidate`, except that it can only be enforced on shared caches and is ignored by private unshared caches. When using this directive, the `public` cache directive must also be used.
`max-age`	Clients can accept a response whose age is no greater that the specified number of seconds.
`s-max-age`	Does the same as `max-age`, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by `s-max-age` overrides the age specified by `max-age`. When using this directive, the `proxy-revalidate` directive must also be used.
`cache-extension`	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

Configuring the Jetty Runtime

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

Namespace

The elements used to configure the Jetty runtime are defined in the namespace `http://cxf.apache.org/transports/http-jetty/configuration`. It is commonly referred to using the prefix httpj. In order to use the Jetty configuration elements you must add the lines shown in Example 40 to the `beans` element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Example 39. 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 9.

Table 9. Elements for Configuring a Jetty Runtime Factory

Element	**Description**
`httpj:engine`	Specifies the configuration for a particular Jetty runtime instance. See The engine element.

`httpj:identifiedTLSServerParameters`	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, `id`, that specifies a unique identifier by which the property set can be referred.
`httpj:identifiedThreadingParameters`	Specifies a reusable set of properties for controlling a Jetty instance's thread pool. It has a single attribute, `id`, that specifies a unique identifier by which the property set can be referred. See Configuring the thread pool.

The engine element

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

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

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

The child elements used to provide the configuration properties are described in Table 10.

Table 10. 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.
`httpj:tlsServerParametersRef`	Refers to a set of security properties defined by an `identifiedTLSServerParameters` element. The `id` attribute provides the `id` of the referred `identifiedTLSServerParameters` element.

Element	Description
`httpj:threadingParameters`	Specifies the size of the thread pool used by the specific Jetty instance. See Configuring the thread poolbookmark179.
`httpj:threadingParametersRef`	Refers to a set of properties defined by a `identifiedThreadingParameters` element. The `id` attribute provides the `id` of the referred `identifiedThreadingParameters` element.

Configuring the thread pool

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

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

The `threadingParameters` has two attributes to specify the size of a thread pool. The attributes are described in Table 11.

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

Table 11. 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 41 shows a configuration fragment that configures a Jetty instance on port number 9001.

Example 40. 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

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 provider sends the consumer a `202 Accepted` response to the consumer over the back-channel of the HTTP connection on which the request was received. It then processes the request and sends the response back to the consumer using a new decoupled server->client HTTP connection. The consumer runtime receives the incoming response and correlates it with the appropriate request before returning to the application code.

Configuring decoupled interactions

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

1. Configure the consumer to use WS-Addressing.

   See Configuring an endpoint to use WS-Addressing.

2. Configure the consumer to use a decoupled endpoint.

   See Configuring the consumer.

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

   See Configuring an endpoint to use WS-Addressing.

Configuring an endpoint to use WS-Addressing

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

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

- Adding the `wswa:UsingAddressing` element to the endpoint's WSDL port element as shown in Example 42.

Example 41. 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 43.

Example 42. Activating WS-Addressing using a Policy

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

```
      <wsam:Addressing
      xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
    <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
  </port>
</service>
...
```

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

Configuring the consumer

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

Example 43 shows the configuration for setting up the endpoint defined in Example 41 to use a decoupled endpoint. The consumer now receives all responses at `http://widgetvendor.net/widgetSellerInbox`.

Example 43. Configuring a Consumer to Use a Decoupled HTTP Endpoint

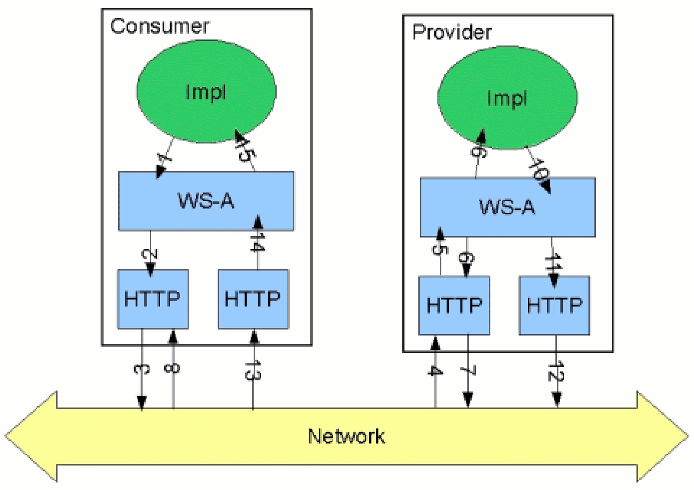
```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http="http://cxf.apache.org/transports/http/conf
    iguration"
    xsi:schemaLocation="http://cxf.apache.org/transports/h
    ttp/configuration
        http://cxf.apache.org/schemas/configuration/http-
      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.h
  ttp-conduit">
    <http:client
    DecoupledEndpoint="http://widgetvendor.net:9999/decoup
    led_endpoint" />
  </http:conduit>
</beans>
```

How messages are processed

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

Figure 1 shows the flow of messages when using HTTP in decoupled mode.

Figure 1. Message Flow in for a Decoupled HTTP Transport



A request starts the following process:

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

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

6. Because the WS-A ReplyTo header is not set to anonymous, the provider sends back a message with the HTTP status code set to `202`, acknowledging that the request has been received.

7. The HTTP layer sends a `202 Accepted` message back to the consumer using the original connection's back-channel.

8. The consumer receives the `202 Accepted` reply on the back-channel of the HTTP connection used to send the original message.

   When the consumer receives the `202 Accepted` reply, the HTTP connection closes.

9. The request is passed to the service provider's implementation where the request is processed.

10. When the response is ready, it is dispatched to the WS-A layer.

11. The WS-A layer adds the WS-Addressing headers to the response message.

12. The HTTP transport sends the response to the consumer's decoupled endpoint.

13. The consumer's decoupled endpoint receives the response from the service provider.

14. The response is dispatched to the consumer's WS-A layer where it is correlated to the proper request using the WS-A RelatesTo header.

15. The correlated response is returned to the client implementation and the invoking call is unblocked.

Using JMS

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

Namespaces

WSDL Namespace

The WSDL extensions used to define a JMS endpoint are specified in the namespace `http://www.w3.org/2010/soapjms/`. In order to use the JMS extensions you will need to add the line shown in Example 45 to the definitions element of your contract.

Example 44. JMS Extension Namespace

```
xmlns:soapjms="http://www.w3.org/2010/soapjms/"
```

Configuration Namespace

The Artix ESB JMS endpoint configuration properties are specified under the `http://www.w3.org/2010/soapjms/` namespace. In order to use the JMS configuration properties you will need to add the line shown in Example 46 to the beans element of your configuration.

Example 45. JMS Configuration Namespaces

```
xmlns:soapjms="http://www.w3.org/2010/soapjms/"
```

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:

- Configuration
- WSDL

Using Configuration

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

NOTE: Information in the configuration file will override the information in the endpoint's WSDL file.

Standard JMS transport configuration in the runtime can be done via Spring dependency injection. Additionally the configuration offers many more options. The connection factory can be resolved via Spring configuration instead of JNDI.

The following example configs use the p-namespace from Spring 2.5. Inside a features element the JMSConfigFeature can be defined.

```
<jaxws:client id="CustomerService"
    xmlns:customer="http://customerservice.example.com/"
    serviceName="customer:CustomerServiceService"
    endpointName="customer:CustomerServiceEndpoint" address="jms://"
    serviceClass="com.example.customerservice.CustomerService">
    <jaxws:features>
        <bean xmlns="http://www.springframework.org/schema/beans"
            class="org.apache.cxf.transport.jms.JMSConfigFeature"
            p:jmsConfig-ref="jmsConfig"/>
    </jaxws:features>
</jaxws:client>
```

The above example references a bean "`jmsConfig`" where the whole configuration for the JMS transport can be done.

A JAX-WS Endpoint can be defined in the same way:

```
<jaxws:endpoint
    xmlns:customer="http://customerservice.example.com/"
    id="CustomerService"
    address="jms://"
    serviceName="customer:CustomerServiceService"
    endpointName="customer:CustomerServiceEndpoint"
    implementor="com.example.customerservice.impl.CustomerServiceImpl">
    <jaxws:features>
        <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
            p:jmsConfig-ref="jmsConfig" />
    </jaxws:features>
</jaxws:endpoint>
```

The JMSConfiguration bean needs at least a reference to a connection factory and a target destination:

```
<bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
    p:connectionFactory-ref="jmsConnectionFactory"
    p:targetDestination="test.cxf.jmstransport.queue"
/>
```

If your ConnectionFactory does not cache connections you should wrap it in a spring `SingleConnectionFactory`. This is necessary because the JMS Transport creates a new connection for each message and the SingleConnectionFactory is needed to cache this connection.

```
<bean id="jmsConnectionFactory"
class="org.springframework.jms.connection.SingleConnectionFactory">
    <property name="targetConnectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
    </property>
</bean>
```

The attributes described in Table 25 configure the connection to the JMS broker.

Table 25. JMS Endpoint Attributes

Attribute	Description
`connectionFactory`	Mandatory field. A reference to a bean that defines a jms ConnectionFactory. Remember to wrap the `connectionFactory` as described above when not using a pooling `ConnectionFactory`.
`reconnectOnException`	If wrapping the `connectionFactory` with a Spring `SingleConnectionFactory` and `reconnectOnException` is `true`, will create a new connection if there is an exception thrown, otherwise will not try to reconnect if the there is an exception thrown. Default is `false`. This option is deprecated. The runtime always reconnects on exceptions.
`targetDestination`	A JNDI name or provider-specific name of a destination. For example, for ActiveMQ: `test.cxf.jmstransport.queue`
`destinationResolver`	A reference to a Spring DestinationResolver. This allows you to define how destination names are resolved to JMS Destinations. By default a `DynamicDestinationResolver` is used. It resolves destinations using the JMS provider’s features. If you reference a `JndiDestinationResolver` you can resolve the destination names using JNDI.
`transactionManager`	A reference to a Spring transaction manager. This allows your webservice to take part in JTA Transactions. You can also register a Spring JMS Transaction Manager to have local transactions.
`pubSubNoLocal`	If `true`, do not receive your own messages when using topics. Default is `false`.

Attribute	Description
`receiveTimeout`	How many milliseconds to wait for a response messages. The default value is changed to `60000` (60 seconds)
`explicitQosEnabled`	If `true`, means that QoS parameters are set for each message. Default is `false`.
`deliveryMode`	`NON_PERSISTENT = 1`. Messages will be kept only in memory `PERSISTENT = 2` (default). Messages will be persisted to disk
`priority`	Priority for the messages. Default is `4`. See your JMS provider documentation for details.
`timeToLive`	After this time, the message will be discarded by the JMS provider. (Default is `0`)
`sessionTransacted`	If `true`, means JMS transactions are used. (Default is `false`).
`messageSelector`	A JMS selector to filter incoming messages (allows shariing a queue).
`subscriptionDurable`	A `durableSubscriptionName`. Default is `false`.
`messageType`	One of `text` (default), `binary`, `byte`.
`pubSubDomain`	`false` (default) means use queues. true means use topics.
`maxSuspendedContinuations`	The maximum suspended continuations that the JMS destination could have. If the current suspended continuations number exceeds this maximum value, the `JMSListenerContainer` will be stopped. The default value is `-1`, which means that this feature is disabled.
`reconnectPercentOfMax`	If the `JMSListenerContainer` is stopped because the number of current suspended continuations exceeds the `maxSuspendedContinuations` value, then the `JMSListenerContainer` will be restarted when the current suspended continuations number falls below the value of (`maxSuspendedContinuations* reconnectPercentOfMax/100`). The default value is `70`.
`createSecurityContext`	Setting this to `true` (default) means create a user security context for incoming messages.

Example

Example 47 shows an Artix ESB configuration entry for configuring the addressing information for a JMS consumer endpoint.

Example 46. 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:p="http://www.springframework.org/schema/p"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd"
       >

    <jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"
              serviceClass="com.example.customerservice.CustomerService">
        <jaxws:features>
            <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
                  p:jmsConfig-ref="jmsConfig"/>
        </jaxws:features>
    </jaxws:client>

    <bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
          p:connectionFactory-ref="jmsConnectionFactory"
          p:targetDestination="test.cxf.jmstransport.queue"
    />

    <bean id="jmsConnectionFactory"
class="org.springframework.jms.connection.SingleConnectionFactory">
        <property name="targetConnectionFactory">
            <bean class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="tcp://localhost:61616" />
            </bean>
        </property>
    </bean>
</beans>
```

Using WSDL

If you prefer to configure your endpoint using WSDL, you can specify JMS endpoints as a part of a WSDL service definition.

NOTE: Information in the configuration file will override the information in the endpoint's WSDL file.

Example

Example 47 shows an example of a JMS WSDL `port` specification.

Example 47. JMS WSDL Port Specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address
location="jms:jndi:dynamicQueues/test.Celtix.jmstransport?jndiURL=
          tcp://localhost:61616&amp;jndiInitialContextFactory=
          org.activemq.jndi.ActiveMQInitialContextFactory"/>
  </port>
</service>
```

Using a named reply destination

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

Setting the reply destination name

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

Example

Example 48 shows the configuration for a JMS client endpoint.

Example 48. JMS Consumer Specification Using a Named Reply Queue

```
<jaxws:client id="CustomerService"
address="jms:jndimyDestination?jndiURL=tcp://localhost:61616&amp;
jndiInitialContextFactory=org.apache.cxf.transport.jms.MyInitialContextFactory&amp;
jndiConnectionFactoryName=myConnectionFactory&amp;
replyToName=myReplyDestination"
                  serviceClass="com.example.customerservice.CustomerService">
    </jaxws:client>
```

Consumer Endpoint Configuration

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS `ByteMessage` or a JMS `TextMessage`. When using an `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 13. messageType Values

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

Example

Example 49 shows a configuration entry for configuring a JMS consumer endpoint.

Example 49. 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:p="http://www.springframework.org/schema/p"
       xmlns:jms="http://cxf.apache.org/transports/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
      http://cxf.apache.org/jaxws
      http://cxf.apache.org/schemas/jaxws.xsd"
       >

    <jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              ...
        <jaxws:features>
            <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
                  p:jmsConfig-ref="jmsConfig"/>
        </jaxws:features>
    </jaxws:client>

    <bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
          p:connectionFactory-ref="jmsConnectionFactory"
          ...
          p:messageType="binary"
          ...
    />

    <bean id="jmsConnectionFactory"
      class="org.springframework.jms.connection.SingleConnectionFactory">
        ....
    </bean>
</beans>
```

Using WSDL

Spcifying the message type

The type of messages accepted by a JMS consumer endpoint is configured by adding the property to the JMS URI that is defined in the <`soap:address`> for our service definition.

Table 14. JMS Client WSDL Extensions

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

Example

Example 50 shows the WSDL for configuring a JMS consumer endpoint.

Example 50. WSDL for a JMS Consumer Endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location=
"jms:jndi:dynamicQueues/test.Celtix.jmstransport?jndiURL=tcp://localhost:61616&amp;
jndiInitialContextFactory=org.activemq.jndi.ActiveMQInitialContextFactory&amp;
messageType=binary"/>
  </port>
</service>
```

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 configured in one of two ways:

- Configuration
- WSDL

NOTE: 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 following WSDL JMA parameters:

Table 15. Provider Endpoint Configuration

Attribute	Description

useConduitSelector	Each conduit is assigned a UUID. If set to `true` this conduit id will be the prefix for all correlation ids. This allows several endpoints to share a JMS queue or topic.
durableSubscriptionName	Specifies the name used to register a durable subscription.
conduitIDSelectorPrefix	If set, this string will be the prefix for all correlation ids that the conduit creates and will also be used in the selector for listening to replies.
sessionTransacted	Set to `true` for resource local transactions. Do not set if you use JTA.

Example

Example 51 shows a Artix ESB configuration entry for configuring a provider endpoint.

Example 51. Configuration for a Provider Endpoint

```
<bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
        p:connectionFactory-ref="jmsConnectionFactory"
        ...
        p:conduitSelectorPrefix="cxf_message_selector"
        p:durableSubscriptionName="cxf subscriber"
        p:useConduitIdSelector="true"
        p:sessionTransacted="true"
        ...
/>
```

Using WSDL

Configuring the endpoint

Provider endpoint behaviors are configured using the same JMS properties as in Table 15.

Example

Example 52 shows the WSDL for configuring a JMS provider endpoint.

Example 52. WSDL for a JMS Provider Endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location=
"jms:jndi:dynamicQueues/test.Celtix.jmstransport?jndiURL=
tcp://localhost:61616&amp;jndiInitialContextFactory=
org.activemq.jndi.ActiveMQInitialContextFactory&amp;sessionTransacted=
true&amp;conduitIdSelectorPrefix=cxf_message_selector&amp;useConduitIdSelector=
true&amp;durableSubscriptionName=cxf_subscriber"/>
  </port>
```

```
</service>
```

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 two types of runtime configuration:

- Consumer specific configuration
- Provider specific configuration

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 can configure consumer runtime behavior using two WSDL JMA parameters 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 parameter `receiveTimeout` in the WSDL. The default timeout interval is `60000`.

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 parameter `timeToLive` in the WSDL. The default time to live interval is 0 which specifies that the request has an infinite time to live.

Example

Example 53 shows a configuration fragment that sets the consumer endpoint's request lifetime to 500 milliseconds and its timeout value to 500 milliseconds.

Example 53. JMS Consumer Endpoint Runtime Configuration

```
...
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
```

```
    <soap:address
location="jms:jndi:dynamicQueues/test.Celtix.jmstransport?jndiURL
=tcp://localhost:61616&amp;jndiInitialContextFactory=org.activemq
.jndi.ActiveMQInitialContextFactory&amp;timeToLive=500&amp;receiv
eTimeout=500"/>
  </port>
</service>
```

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.

Configuring the durable subscriber identifier

The WSDL JMS parameter `durableSubscriptionClientId` specifies the client identifier the endpoint uses to create and access durable subscriptions.

Example

Example 54 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 54. Provider Endpoint Runtime Configuration

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location=
"jms:jndi:dynamicQueues/test.Celtix.jmstransport?jndiURL=
tcp://localhost:61616&amp;
jndiInitialContextFactory=
org.activemq.jndi.ActiveMQInitialContextFactory&amp;
timeToLive=500&amp;
durableSubscriptionClientId=jms-test-id"/>
  </port>
</service>
```

Using WebSphere MQ

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

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.

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 WSDL JMS parameters to specify the correct JMS URI that the JMS provider will use. Table 18 shows the attributes needed when using WebSphere MQ's JMS interface.

Table 18. 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.
`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 55, 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 55. Specifying the JNDI Initial Context Factory

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location=
"jms:jndi:dynamicQueues/test.Celtix.jmstransport?jndiURL=tcp://localhost:1414&amp;
jndiInitialContextFactory=com.ibm.mq.jms.context.WMQInitialContextFactory"/>
  </port>
</service>
```

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 the `jndiURL` property to the `<soap:address>` element in the wsdl or spring configuration file. See Example 55 for the use of this property in the WSDL.

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

- The default WebSphere MQ installation includes JNDI providers for local file systems 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.