Artix 5.6.3

Bindings and Transports, C++ Runtime

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

http://www.microfocus.com

2015-02-24

Contents

Preface

What is Covered in This Book

This book discusses the bindings and transports supported by Artix ESB for C++. 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. In the case of transports, such as WebSphere MQ, it also discusses how to access some of the transports more advanced features.

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 Artix ESB for C++. 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 three parts:

- Part I describes how to work with the message bindings.
- Part II describes how to work with the transports.
- Part III describes how to use other Artix ESB features that are contract driven. This includes codeset conversion and XSLT.

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/orbix/orbix-6.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

Artix ESB for C++ supports the following bindings:

- SOAP
- CORBA
- Fixed record length
- Pure XML
- Tagged (variable record length)
- Tuxedo's Field Manipulation Language (FML)

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 portType} {-n namespace} [-b binding] [-d dir]
[-o file] [-style {[document] | [rpc]}] [-use {[literal] |
[encoded]}] [-quiet] [-verbose] [-h] [-v] wsdl_file
```

The command has the following options:

Option	**Description**
`-i` *`portType`*	Specifies the name of the port type being mapped to a SOAP 1.1 binding.
`-n` *`namespace`*	Specifies the namespace to use for the SOAP 1.1 binding.
`-b` *`binding`*	Specifies the name for the generated SOAP binding. Defaults to *`portType`*`Binding`.
`-d` *`dir`*	Specifies the directory into which the new WSDL file is written.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to `wsdl_file-soap.wsdl`.
`-style`	Specifies the encoding style to use in the SOAP binding. Defaults to document.
`-use`	Specifies how the data is encoded. Default is `literal`.
`-quiet`	Specifies that the tool runs in quiet mode.

Option	Description
-verbose	Specifies that the tool runs in verbose mode.
-h	Specifies that the tool will display a usage message.
-v	Displays the tool’s version.

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

Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in an Artix contract similar to the one 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"/>
```

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 into the SOAP header must be defined using an element. Together the `message` and the `part` attributes fully describe the data to insert into the SOAP header.

Example 3. SOAP Header Syntax

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

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

Splitting messages between body and header

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

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

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

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/widgetTyp
         es" 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. However, you can add SOAP headers to your binding using any text or XML editor.

Adding a SOAP 1.2 Binding

Artix provides the wsdltosoap 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:

`wsdltosoap` {–soapversion 1.2} {-i *`portType`*} {-n *`namespace`*} [-b *`binding`*] [-d *`dir`*] [-o *`file`*] [-style [[document] | [rpc]]] [-use {[literal] | [encoded]}] [-quiet] [-verbose] [-h] [-v] *`wsdl_file`*

The command has the following options:

Option	Description
`–soapversion 1.2`	Specifies that the generated binding should use SOAP 1.2.
`-i` *`portType`*	Specifies the name of the port type being mapped to a SOAP binding.
`-n` *`namespace`*	Specifies the namespace to use for the SOAP binding.
`-b` *`binding`*	Specifies the name for the generated SOAP binding. Defaults to *`portType`*Binding.
`-d` *`dir`*	Specifies the directory into which the new WSDL file is written.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl_file`*`-soap.wsdl`.
`-style`	Specifies the encoding style to use in the SOAP binding. Defaults to `document`.

Option	Description
`-use`	Specifies how the data is encoded. Default is `literal`.
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.
`-h`	Specifies that the tool will display a usage message.
`-v`	Displays the tool’s version’

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

Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in an Artix contract similar to the one shown in Example 6 on page 45.

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:wsoap12="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">
  <wsoap12:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
              <operation name="placeWidgetOrder">
     <wsoap12:operation soapAction="" style="document"/>
     <input name="order">
                <wsoap12:body use="literal"/>
     </input>
     <output name="bill">
                <wsoap12:body use="literal"/>
     </output>
     <fault name="sizeFault">
                <wsoap12: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 SOAP headers 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="document"/>
     <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.

Attribute	Description
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.
`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 on page 54 and Step 5 on page 54.

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 part, `imageData`. The remaining two parts of the input message, `patientName` and `patientNumber`, are sent in the first part of the MIME multipart image as part of the SOAP body.

Example 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.
2. Enable the runtime's MTOM support.

Defining Data Types to use MTOM

When defining a data type for passing along a block of binary data, such as an image file or a sound file, in WSDL you define the element for the data to be of type xsd:base64Binary or of type xsd:hexBinary. By default, any element of either type results in the generation of a C++ type which can be serialized using MTOM.

You can also specify the MIME type of the binary data using the `xmime:contentType` attribute or `xmime:expectedContentType` attributes. In addition, you can define the types of the elements binary data using xmime versions of the binary data types. When you use the MIME data to an element definition, Artix generates special data types to contain the data.

Using the XML Schema binary types

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 for sending along as part of a normal SOAP message.

Example 13. Message for MTOM Using XML Schema Types

```
<?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>
```

The data contained in the `imageData` element of the request message will be sent using MTOM if the consumer is properly configured. The runtime will automatically set the XOP package's `Content-Type` property to `application/octet-stream`.

Using XMime attributes to specifying the content type

Some Java Web Service frameworks use MIME type information to optimize how their runtime manages binary data. This is done using the `xmime:expectedContentTypes` attribute. This attribute is defined in the http://www.w3.org/2005/05/xmlmime namespace. It is placed on the element's schema type definition and specifies the MIME types that the element is expected to contain. The `xmime:expectedContentTypes` attribute takes a comma separated list of MIME types.

When the C++ code generator sees these attributes on an element it generates Artix-specific objects to hold the data:

- If the element was defined using xsd:base64Binary the code generator will create an `IT_Bus::XMimeBase64Binary` object for it.

- If the element was defined using xsd:hexBinary the code generator will create an `IT_Bus::XMimeHexBinary` object for it.

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

Example 14 shows how you would modify xRayType from Example 13 to specify the type of content passed in the binary data field.

Example 14. Using XMime Attributes to Specify the Contents of a Binary Field

```
...
 <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="image/gif"/>
       </sequence>
     </complexType>
     <element name="xRay" type="xsd1:xRayType" />
   </schema>
 </types>
...
```

Using the XMime binary types

An alternate approach to specifying the data type of elements that will be passed using MTOM is to define them using the MIME binary types. The MIME binary types, xmime:base64Binary and xmime:hexBinary are defined in the http://www.w3.org/2005/05/xmlmime namespace. They can be used as direct replacements for their respective XML Schema data types.

Example 15 shows the xRayType, from Example 13, defined using the MIME binary types.

Example 15. Message for MTOM Using MIME Binary Types

```
<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="xmime:base64Binary"
/>
     </sequence>
    </complexType>
    <element name="xRay" type="xsd1:xRayType" />
  </schema>
</types>
```

The Artix code generator creates Artix-specific data types for elements defined using the MIME binary types:

- Elements of type `xmime:base64Binary` are mapped to `IT_Bus::XMimeBase64Binary` objects.
- Elements of type `xmime:hexBinary` are mapped to `IT_Bus::XMimeHexBinary` objects.

Enabling MTOM

By default, Artix service consumers do not use MTOM to transmit binary data in SOAP messages. You need to configure them to use this functionality.

Service providers, on the other hand, do not need any special configuration to handle SOAP messages that use MTOM. They can receive and send MTOM data by default.

Service consumers

You control the runtime's MTOM support using the `plugins:soap:enable_mtom` and `plugins:soap12:enable_mtom` configuration variables. These are boolean variables whose default settings are `false`.

To activate the runtime's MTOM support set this variable to `true`.

For more information on configuring Artix ESB for C++, see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

Example 16 shows configuration for an endpoint that is MTOM= enabled.

Example 16. Configuration for Enabling MTOM in SOAP 1.2

```
demos { mtom {
    plugins:soap12:enable_mtom = "true";
  };
};
```

Service providers

Artix SOAP 1.2 service providers do not need to be configured to use MTOM.

When a service provider receives a SOAP 1.2 request that uses MTOM, it will respond using MTOM.

Using Tuxedo's FML Buffers

Artix can send and receive messages packaged as FML buffers.

Tuxedo's native data format is FML. The FML buffers used by Tuxedo applications are described in one of two ways:

- A *field table file* that is loaded at runtime.
- A C header file that is compiled into the application.

A field table file is a detailed and user readable text file describing the contents of a buffer. It clearly describes each field's name, ID number, data type, and a comment. Using the FML library calls, Tuxedo applications map the field table description to usable `fldids` at runtime.

The C header file description of an FML buffer simply maps field names to their `fldid`. The `fldid` is an integer value that represents both the type of data stored in a field and a unique identifying number for that field.

Artix works with this data by mapping the native Tuxedo data descriptions into a WSDL `binding` element. As part of developing an Artix solution to integrate with legacy Tuxedo applications, you must add an FML binding to the contract describing the integration.

FML/XML Schema support

An FML buffer can only contain the data types listed in Table 3.

Table 3. FML Type Support

FML Type	XML Schema Type
short	xsd:short
short	xsd:unsignedShort
long	xsd:int
long	xsd:unsignedInt
float	xsd:float
double	xsd:double
string	xsd:string

FML Type	XML Schema Type
string	xsd:base64Binary
string	xsd:hexBinary

Important: Due to FML limitations, support for complex types is limited to `xsd:sequence` and `xsd:all`.

Mapping from a field table to an Artix contract

Creating an Artix contract to represent an FML buffer is a two-step process:

1. Create the logical data representation of the FML buffer in the Artix contract as described in Mapping to logical type descriptions.
2. Enter the FML binding information using Artix WSDL extensors as described in Adding the FML binding.

Mapping to logical type descriptions

To create a logical data type to represent data in an FML buffer do the following:

1. If the C header file for the FML buffer does not exist, generate it from the field table using the Tuxedo **mkfldhdr** or **mkfldhdr32** utility program.
2. For each field in the FML buffer, create an `element` with the following attribute settings:
   - `name` is set to a string that identifies the field. This value is used to by the binding to correlate the logical type with the FML field.

     TIP: The value of the `name` attribute does not need to match the name of the physical FML field.

   - `type` is set to the appropriate XML Schema type for the type specified in the field table. See FML/XML Schema support.
3. If your Tuxedo application has data fields that are always used together, you can group the corresponding elements into complex types.

Important: In Tuxedo, a WSDL operation is implicitly bound to the Tuxedo service used. So, when the Tuxedo extensor is configured for the WSDL port there must be a one-to-one mapping between the WSDL operation and the Tuxedo service. You should only group elements into complex types only if they appear together in all exposed Tuxedo services.

Consider a Tuxedo application that returns personnel records on employees that needs to be exposed through a new web interface. The Tuxedo application uses the field table file shown in Example 17.

Example 17. `personnelInfo` Field Table File

```
# personnelInfo field
# name      Number     type      flags     comment
empID       100        long      -
name        101        string    -
age         102        short     -
dept        103        string    -
addr        104        string    -
city        105        string    -
state       106        string    -
zip         107        string    -
```

The C++ header file generated by the Tuxedo **mkfldhdr** tool to represent the `personnelInfo` FML buffer is shown in Example 18. Even if you are not planning to access the FML buffer using the compile-time method, you will need to generate the header file when using Artix because this will give you the `fldid` values for the fields in the buffer.

Example 18. `personnelInfo` C++ header

```
/*        fname     fldid                */
/*        -----     -----                */
#define   empID     ((FLDID)8293)     /* number: 100     type: long */
#define   name      ((FLDID)41062)    /* number: 101     type: string */
#define   age       ((FLDID)102)      /* number: 102     type: short */
#define   dept      ((FLDID)41064)    /* number: 103     type: string */
#define   addr      ((FLDID)41065)    /* number: 104     type: string */
#define   city      ((FLDID)41066)    /* number: 105     type: string */
#define   state     ((FLDID)41067)    /* number: 106     type: string */
#define   zip       ((FLDID)41068)    /* number: 107     type: string */
```

Before mapping the FML buffer into your contract, you need to look at the operations exposed by the Tuxedo application. Suppose it exposes two operations:

- `infoByName()` that returns the employee data based on a name search.
- `infoByID()` that returns the employee data based on the employee's ID number.

Because the employee data is always returned as a unit you can group it into a complex type as shown in Example 19.

Example 19. Logical description of personneInfo FML buffer

```
<types>
  <schema targetNamespace="http://soapinterop.org/xsd"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="personnelInfo">
<sequence>
      <element name="empId" type="xsd:int"/>
      <element name="name" type="xsd:string"/>
      <element name="age" type="xsd:short"/>
      <element name="dept" type="xsd:string"/>
      <element name="addr" type="xsd:string"/>
      <element name="city" type="xsd:string"/>
      <element name="state" type="xsd:string"/>
      <element name="zip" type="xsd:string"/>
</sequence>
    </complexType>
    ...
  </schema>
</types>
```

The interface for your Tuxedo application would be mapped to a portType similar to Example 20.

Example 20. personnelInfo Lookup Interface

```
<message name="idLookupRequest">
  <part name="empId" type="xsd:int"/>
</message>
<message name="nameLookupRequest">
  <part name="empId" type="xsd:string"/>
</message>
<message name="lookupResponse">
  <part name="return" element="xsd1:personnelInfo"/>
</message>

<portType name="personelInfoLookup">
  <operation name="infoByName">
    <input name="name" message="nameLookupRequest"/>
    <output name="return" message="lookupResponse"/>
  </operation>
  <operation name="infoByID">
    <input name="id" message="idLookupRequest"/>
    <output name="return" message="lookupResponse"/>
  </operation>
</portType>
```

Flattened XML and FML

While XML Schema allows you to create structured data that is organized in multiple layers, FML data is flat. All of the elements in a field table exist on the same level. To handle this difference Artix flattens out the XML data when it is passed through the FML binding.

As a result, complex types defined in XML Schema are collapsed into their composite elements. For instance, the message lookupResponse, which uses the complex type defined in Example 19, would be equivalent to the message definition in Example 21 when processed by the FML binding.

Example 21. Flattened Message for FML

```
<message name="lookupResponse">
  <part name="empId" type="xsd:int"/>
  <part name="name" type="xsd:string"/>
  <part name="age" type="xsd:short"/>
  <part name="dept" type="xsd:string"/>
  <part name="addr" type="xsd:string"/>
  <part name="city" type="xsd:string"/>
  <part name="state" type="xsd:string"/>
  <part name="zip" type="xsd:string"/>
</message>
```

Adding the FML binding

To add the binding that maps the logical description of the FML buffer to a physical FML binding do the following:

1. Add the following line in the `definition` element at the beginning of the contract.

```
xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
```

2. Create a new `binding` element in your contract to define the FML buffer's binding.

3. Add a `tuxedo:binding` element to identify that this binding defines an FML buffer.

4. Add a `tuxedo:fieldTable` element to the binding to describe how the element names defined in the logical portion of the contract map to the `fldid` values for the corresponding fields in the FML buffer.

   The `tuxedo:fieldTable` has a mandatory `type` attribute. `type` can be either `FML` for specifying that the application uses FML16 buffers or `FML32` for specifying that the application uses FML32 buffers.

5. For each element in the logical data type, add a `tuxedo:field` element to the `tuxedo:fieldTable` element.

   `tuxedo:field` defines how the logical data elements map to the physical FML buffer. It has two mandatory attributes:

Table 4. Attributes of the FML Binding's Field Element

Attribute	Description
name	Specifies the name of the element or message part that describes the field.
id	Specifies the fldid value for the field in the FML buffer

6. For each operation in the interface, create a standard WSDL `operation` element to define the operation being bound.
7. For each operation, add a standard WSDL `input` and `output` elements to the `operation` element to define the messages used by the operation.
8. For each operation, add a `tuxedo:operation` element to the `operation` element.

For example, the binding for the `personalInfo` FML buffer, defined in Example 17, will be similar to the binding shown in Example 22.

Example 22. `personalInfo` FML binding

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="personalInfoService" targetNamespace=
"http://info.org/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://soapinterop.org/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
   >
...
 <binding name="personalInfoFMLBinding"
type="tns:personnalInfoLookup">
   <tuxedo:binding/>
   <tuxedo:fieldTable type="FML">
     <tuxedo:field name="empId" id="8293"/>
     <tuxedo:field name="name" id="41062"/>
     <tuxedo:field name="age" id="102"/>
     <tuxedo:field name="dept" id="41064"/>
     <tuxedo:field name="addr" id="41065"/>
     <tuxedo:field name="city" id="41066"/>
     <tuxedo:field name="state" id="41067"/>
     <tuxedo:field name="zip" id="41068"/>
   </fml:idNameMapping>
   <operation name="infoByName">
     <tuxedo:operation/>
     <input name="name"/>
     <output name="return"/>
   </operation>
   <operation name="infoByName">
```

```
      <tuxedo:operation/>
      <input name="name"/>
      <output name="return"/>
    </operation>
  </binding>
...
</definitions>
```

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.

You can create an XML binding using any text or XML editor.

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 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 binding namespace

The extensions used to describe XML format bindings are defined in the namespace `http://celtix.objectweb.org/bindings/xmlformat`. Artix ESB tools use the prefix xformat to represent the XML binding extensions. Add the following line to your contracts:

```
xmlns:xformat="http://celtix.objectweb.org/bindings/xmlformat"
```

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 23. 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 24. 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 25. 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 26. 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"/>
```

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 27. XML Document generated using the rootNode attribute

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

Overriding the binding's rootNode attribute setting

You can also set the `rootNode` attribute for each individual message, or override the global setting for a particular message, by using the `xformat:body` element inside of the message binding. For example, if you wanted the output message defined in Example 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 28. 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>
```

Using Fixed Length Records

To make interoperating with mainframes and older systems easy, Artix ESB can send and receive messages formatted as fixed length records.

The Artix ESB fixed binding is used to represent fixed record length data.

Common uses for this type of payload format are communicating with back-end services on mainframes and applications written in COBOL. Artix ESB provides several means for creating a contract containing a fixed binding:

- If you are integrating with an application written in COBOL and have the COBOL copybook defining the data to be used, you can import the copybook to create a contract.
- If you have a description of the fixed data in some form other than a COBOL copybook, you can create a contract by describing the data.
- You can enter the binding information using any text editor or XML editor.

Hand editing

To map a logical interface to a fixed binding you do the following:

1. Add the proper namespace reference to the `definition` element of your contract. See Fixed binding namespace.
2. Add a WSDL `binding` element to your contract to hold the fixed binding, give the binding a unique `name`, and specify the port type that represents the interface being bound.
3. Add a `fixed:binding` element as a child of the new `binding` element to identify this as a fixed binding and set the element's attributes to properly configure the binding. See fixed:binding.
4. For each operation defined in the bound interface, add a WSDL `operation` element to hold the binding information for the operation's messages.
5. For each operation added to the binding, add a `fixed:operation` child element to the `operation` element. See fixed:operation.

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. For each `input`, `output`, and `fault` element in the binding, add a `fixed:body` child element to define how the message parts are mapped into the concrete fixed record length payload. See fixed:body.

Fixed binding namespace

The extensions used to describe fixed record length bindings are defined in the namespace `http://schemas.iona.com/bindings/fixed`. Artix tools use the prefix fixed to represent the fixed record length extensions. Add the following line to your contract:

```
xmlns:fixed="http://schemas.iona.com/bindings/fixed
```

fixed:binding

`fixed:binding` specifies that the binding is for fixed record length data. Its attributes are described in Table 5.

Table 5. Attributes for `fixed:binding`

Attribute	Description
`justification`	Specifies the default justification of the data contained in the messages. Valid values are `left` and `right`. Default is `left`.
`encoding`	Specifies the codeset used to encode the text data. Valid values are any valid ISO locale or Internet Assigned Numbers Authority (IANA) codeset name. Default is `UTF-8`.
`padHexCode`	Specifies the hex value of the character used to pad the record.

The settings for the attributes on these elements become the default settings for all the messages being mapped to the current binding. All of the values can be overridden on a message-by-message basis.

fixed:operation

`fixed:operation` is a child element of the WSDL `operation` element and specifies that the operation's messages are being mapped to fixed record length data.

`fixed:operation` has one attribute, `discriminator`, that assigns a unique identifier to the operation. If your service only defines a single operation, you do not need to provide a discriminator. However, if your service has more than one service, you must define a unique discriminator for each operation in the service. Not doing so will result in unpredictable behavior when the service is deployed.

For each message used in the operation, you will need to include a fixed:field element whose `name` attribute is equal to the value of `discriminator` and whose `bindingOnly` attribute is set to `true`. This field will hold the value used by the binding to discriminate between the operations. For more information see fixed:field.

fixed:body

`fixed:body` is a child element of the `input`, `output`, and `fault` elements representing the messages being mapped to fixed record length data. It specifies that the message body is mapped to fixed record length data on the wire and describes the exact mapping for the message's parts.

To fully describe how a message is mapped into the fixed message do the following:

1. If the default justification, padding, or encoding settings for the attribute are not correct for this particular message, override them by setting the following optional attributes for `fixed:body`.

Table 6. Attributes for `fixed:body`

Attribute	Description
`justification`	Specifies how the data in the messages are justified. Valid values are `left` and `right`.
`encoding`	Specifies the codeset used to encode text data. Valid values are any valid ISO locale or IANA codeset name.
`padHexCode`	Specifies the hex value of the character used to pad the record.

2. For each part in the message the `fixed:body` element is binding, add the appropriate child element to define the part's concrete format on the wire.

   The following child elements are used in defining how logical data is mapped to a concrete fixed format message:

   `fixed:field` maps message parts defined using a simple type. See the *XML Schema Simple Types* section in ***Writing Artix ESB Contracts***.

   `fixed:sequence` maps message parts defined using a sequence complex type. Complex types defined using `all` are not supported by the fixed format binding. See the *Defining Data Structures* section in ***Writing Artix ESB Contracts***.

   `fixed:choice` maps message parts defined using a choice complex type. See the *Defining Data Structures section* in ***Writing Artix ESB Contracts***.

3. If you need to add any fields that are specific to the binding and that will not be passed to the applications, define them using a `fixed:field` element with its `bindingOnly` attribute set to `true`.

   When `bindingOnly` is set to `true`, the field described by the `fixed:field` element is not propagated beyond the binding. For input messages, this means that the field is read in and then discarded. For output messages, you must also use the `fixedValue` attribute.

The order in which the message parts are listed in the `fixed:body` element represent the order in which they are placed on the wire. It does not need to correspond to the order in which they are specified in the `message` element defining the logical message.

fixed:field

`fixed:field` is used to map simple data types to a fixed length record. To define how the logical data is mapped to a fixed field do the following:

1. Create a `fixed:field` child element to the `fixed:body` element representing the message.

2. Set the `fixed:field` element's `name` attribute to the name of the message part defined in the logical message description that this element is mapping.

3. If the data being mapped is of type xsd:string, a simple type that has xsd:string as its base type, or an enumerated type set, the `size` attribute of the `fixed:field` element.

NOTE: If the message part is going to hold a date you can opt to use the `format` attribute described in Step 4 on page 80 instead of the `size` attribute.

The `size` attribute specifies the length of the string record in the concrete fixed message. For example, the logical message part, `raverID`, described in Example 29 would be mapped to a `fixed:field` similar to Example 30.

Example 29. Fixed String Message

```
<message name="fixedStringMessage">
 <part name="raverID" type="xsd:string"/>
</message>
```

In order to complete the mapping, you must know the length of the record field and supply it. In this case, the field, `raverID`, can contain no more than twenty characters.

Example 30. Fixed String Mapping

```
<fixed:field name="raverID" size="20"/>
```

4. If the data being mapped is of a numerical type, like xsd:int, or a simple type that has a numerical type as its base type, set the `fixed:field` element's `format` attribute.

The `format` attribute specifies how non-string data is formatted. For example, if a field contains a 2-digit numeric value with one decimal place, it would be described in the logical part of the contract as an xsd:float, as shown in Example 31.

Example 31. Fixed Record Numeric Message

```
<message name="fixedNumberMessage">
  <part name="rageLevel" type="xsd:float"/>
</message>
```

From the logical description of the message, Artix has no way of determining that the value of `rageLevel` is a 2-digit number with one decimal place because the fixed record length binding treats all data as characters. When mapping `rageLevel` in the fixed binding the value of the `format` attribute would be ##.#, as shown in Example 32. This provides Artix with the meta-data needed to properly handle the data.

Example 32. Mapping Numerical Data to a Fixed Binding

```
<fixed:field name="rageLevel" format="##.#"/>
```

Dates are specified in a similar fashion. For example, the value of the `format` attribute for the date `12/02/72` is `MM/DD/YY`. When using the fixed binding it is recommended that dates are described in the logical part of the contract using xsd:string. For example, a message containing a date would be described in the logical part of the contract as shown in Example 33.

Example 33. Fixed Date Message

```
<message name="fixedDateMessage">
  <part name="goDate" type="xsd:string"/>
</message>
```

If `goDate` is entered using the standard short date format for US English locales, `mm/dd/yyyy`, you would map it to a fixed record field as shown in Example 34.

Example 34. Fixed Format Date Mapping

```
<fixed:field name="goDate" format="mm/dd/yyyy"/>
```

5. If the justification setting is not correct for this particular field, override it by setting the `justification` attribute. Valid values are `left` and `right`.

6. If you want the message part to have a fixed value no matter what data is set in the message part by the application, set the `fixed:field` element's `fixedValue` attribute instead of the `size` or the `format` attribute.

   The `fixedValue` attribute specifies a static value to be passed on the wire. When used without `bindingOnly="true"`, the value specified by the `fixedValue` attribute replaces any data that is stored in the message part passed to the fixed record binding. For example, if goDate, shown in Example 33, were mapped to the fixed field shown in Example 35 the actual message returned from the binding would always have the date 11/11/2112.

Example 35. `fixedValue` Mapping

```
<fixed:field name="goDate" fixedValue="11/11/2112"/>
```

7. If the data being mapped is of an enumerated type, see "Defining Enumerated Types" on page 43, add a `fixed:enumeration` child element to the `fixed:field` element for each possible value of the enumerated type.

fixed:enumeration takes two required attributes: value and fixedValue. The value attribute corresponds to the enumeration value as specified in the logical description of the enumerated type. The fixedValue attribute specifies the concrete value that will be used to represent the logical value on the wire.

For example, if you had an enumerated type with the values FruityTooty, Rainbow, BerryBomb, and OrangeTango the logical description of the type would be similar to Example 36.

Example 36. Enumeration Logical Mapping

```
<xs:simpleType name="flavorType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="FruityTooty"/>
    <xs:enumeration value="Rainbow"/>
    <xs:enumeration value="BerryBomb"/>
    <xs:enumeration value="OrangeTango"/>
  </xs:restriction>
```

When you map the enumerated type, you need to know the concrete representation for each of the enumerated values. The concrete representations can be identical to the logical or some other value. The enumerated type in Example 36 could be mapped to the fixed field shown in Example 37. Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to OrangeTango.

Example 37. Fixed Ice Cream Mapping

```
<fixed:field name="flavor" size="2">
  <fixed:enumeration value="FruityTooty"
                     fixedValue="FT"/>
  <fixed:enumeration value="Rainbow" fixedValue="RB"/>
  <fixed:enumeration value="BerryBomb"
                     fixedValue="BB"/>
  <fixed:enumeration value="OrangeTango"
                     fixedValue="OT"/>
</fixed:field>
```

Note that the parent fixed:field element uses the size attribute to specify that the concrete representation is two characters long. When mapping enumerations, the size attribute will always be used to represent the size of the concrete representation.

fixed:choice

`fixed:choice` is used to map choice complex types into fixed record length messages. To map a choice complex type to a `fixed:choice` do the following:

1. Add a `fixed:choice` child element to the `fixed:body` element.

2. Set the `fixed:choice` element's `name` attribute to the name of the logical message part being mapped.

3. Set the `fixed:choice` element's optional `discriminatorName` attribute to the name of the field used as the discriminator for the union.

   The value for `discriminatorName` corresponds to the name of a binding only `fixed:field` element that describes the type used for the union's discriminator as shown in Example 38. The only restriction in describing the discriminator is that it must be able to handle the values used to determine the case of the union. Therefore the values used in the union mapped in Example 38 must be two-digit integers.

Example 38. Using the `discriminatorName` Attribute

```
<fixed:field name="disc" format="##"
          bindingOnly="true"/>
<fixed:choice name="unionStation"
           discriminatorName="disc">
...
</fixed:choice>
```

4. For each element in the logical definition of the message part, add a `fixed:case` child element to the `fixed:choice` element.

fixed:case

`fixed:case` elements describe the complete mapping of a choice complex type element to a fixed record length message. To map a choice complex type element to a `fixed:case` do the following:

1. Set the `fixed:case` element's `name` attribute to the name of the logical definition's element.

2. Set the `fixed:case` element's `fixedValue` attribute to the value of the discriminator that selects this element. The value of the `fixedValue` attribute must correspond to the format specified by the `discriminatorName` attribute of the parent `fixed:choice` element.

3. Add a child element to define how the element's data is mapped into a fixed record.

   The child elements used to map the part's type to the fixed message are the same as the possible child elements of a `fixed:body` element. As with a `fixed:body` element, a `fixed:sequence` is made up of `fixed:field` elements to describe simple types, `fixed:choice` elements to describe choice complex types, and `fixed:sequence` elements to describe sequence complex types.

Example 39 shows an Artix contract fragment mapping a choice complex type to a fixed record length message.

Example 39. Mapping a Union to a Fixed Record Length Message

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
   targetNamespace="http://www.iona.com/FixedService"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:fixed="http://schemas.iona.com/bindings/fixed"
   xmlns:tns="http://www.iona.com/FixedService"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema
        targetNamespace="http://www.iona.com/FixedService"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="unionStationType">
     <xsd:choice>
       <xsd:element name="train" type="xsd:string"/>
       <xsd:element name="bus" type="xsd:int"/>
       <xsd:element name="cab" type="xsd:int"/>
       <xsd:element name="subway" type="xsd:string"/>
     </xsd:choice>
   </xsd:complexType>
...
</types>
<message name="fixedSequence">
  <part name="stationPart" type="tns:unionStationType"/>
</message>
<portType name="fixedSequencePortType">
...
</portType>
<binding name="fixedSequenceBinding"
       type="tns:fixedSequencePortType">
  <fixed:binding/>
...
   <fixed:field name="disc" format="##"
   bindingOnly="true"/>
   <fixed:choice name="stationPart"
descriminatorName="disc">
     <fixed:case name="train" fixedValue="01">
```

```
       <fixed:field name="name" size="20"/>
     </fixed:case>
     <fixed:case name="bus" fixedValue="02">
       <fixed:field name="number" format="###"/>
     </fixed:case>
     <fixed:case name="cab" fixedValue="03">
       <fixed:field name="number" format="###"/>
     </fixed:case>
     <fixed:case name="subway" fixedValue="04">
<fixed:field name="name" format="10"/>
     </fixed:case>
</fixed:choice>
  ...
</binding>
...
</definition>
```

fixed:sequence

fixed:sequence maps sequence complex types to a fixed record length message. To map a sequence complex type to a fixed:sequence do the following:

1. Add a fixed:sequence child element to the fixed:body element.

2. Set the fixed:sequence element's name attribute to the name of the logical message part being mapped.

3. For each element in the logical definition of the message part, add a child element to define the mapping for the part's type to the physical fixed message.

   The child elements used to map the part's type to the fixed message are the same as the possible child elements of a fixed:body element. As with a fixed:body element, a fixed:sequence is made up of fixed:field elements to describe simple types, fixed:choice elements to describe choice complex types, and fixed:sequence elements to describe sequence complex types.

4. If any elements of the logical data definition have occurrence constraints, see Defining Data Structures on page 34, map the element into a fixed:sequence element with its occurs and counterName attributes set.

   The occurs attribute specifies the number of times this sequence occurs in the message buffer. counterName specifies the name of the field used for specifying the number of sequence elements that are actually being sent in the message. The value of counterName corresponds to a binding-only fixed:field with at least enough digits to count to the value specified in occurs as shown in Example 40. The value passed to the counter field can be any number up to the value specified by occurs and allows operations to use less than the specified number of sequence elements. Artix ESB will pad out the sequence to the number of elements specified by occurs when the data is transmitted to the receiver so that the receiver will get the data in the promised fixed format.

Example 40. Using counterName

```
<fixed:field name="count" format="##" bindingOnly="true"/>
<fixed:sequence name="items" counterName="count" occurs="10">
...
</fixed:sequence>
```

For example, a structure containing a name, a date, and an ID number would contain three fixed:field elements to fully describe the mapping of the data to the fixed record message. Example 41 shows an Artix contract fragment for such a mapping.

Example 41. Mapping a Sequence to a Fixed Record Length Message

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
   targetNamespace="http://www.iona.com/FixedService"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:fixed="http://schemas.iona.com/bindings/fixed"
   xmlns:tns="http://www.iona.com/FixedService"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema
        targetNamespace="http://www.iona.com/FixedService"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="person">
<xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="date" type="xsd:string"/>
      <xsd:element name="ID" type="xsd:int"/>
   </xsd:complexType>
...
</types>
<message name="fixedSequence">
  <part name="personPart" type="tns:person"/>
</message>
<portType name="fixedSequencePortType">
...
</portType>
<binding name="fixedSequenceBinding"
type="tns:fixedSequencePortType">
  <fixed:binding/>
...
```

```
    <fixed:sequence name="personPart">
      <fixed:field name="name" size="20"/>
      <fixed:field name="date" format="MM/DD/YY"/>
      <fixed:field name="ID" format="#####"/>
    </fixed:sequence>
...
</binding>
...
</definition>
```

Example

Example 42 shows an example of a contract containing a fixed record length message binding.

Example 42. Fixed Record Length Message Binding

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

<types>
<schema
targetNamespace="http://widgetVendor.com/types/widgetTypes"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

<xsd:simpleType name="widgetSize">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="big"/>
    <xsd:enumeration value="large"/>
    <xsd:enumeration value="mungo"/>
    <xsd:enumeration value="gargantuan"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="Address">
<xsd:sequence>
<xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street1" type="xsd:string"/>
        <xsd:element name="street2" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zipCode" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
     <xsd:complexType name="widgetOrderInfo">
<xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="shippingAddress"
        type="xsd1:Address"/>
</xsd:sequence>
</xsd:complexType>
     <xsd:complexType name="widgetOrderBillInfo">
<xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="amtDue" type="xsd:float"/>
        <xsd:element name="orderNumber" type="xsd:string"/>
        <xsd:element name="shippingAddress"
        type="xsd1:Address"/>
</xsd:sequence>

</xsd:complexType>
   </schema>
  </types>
<message name="widgetOrder">
<part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
  </message>
  <message name="widgetOrderBill">
    <part name="widgetOrderConformation"
    type="xsd1:widgetOrderBillInfo"/>
  </message>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
  </portType>
  <binding name="orderWidgetsBinding"
  type="tns:orderWidgets">
    <fixed:binding/>
      <operation name="placeWidgetOrder">
        <fixed:operation discriminator="widgetDisc"/>
<input name="widgetOrder">
        <fixed:body>
          <fixed:sequence name="widgetOrderForm">
            <fixed:field name="amount" format="###"/>
            <fixed:field name="order_date"
            format="MM/DD/YYYY"/>
            <fixed:field name="type" size="2">
              <fixed:enumeration value="big"
              fixedValue="bg"/>
              <fixed:enumeration value="large"
              fixedValue="lg"/>
              <fixed:enumeration value="mungo"
              fixedValue="mg"/>
              <fixed:enumeration value="gargantuan"
              fixedValue="gg"/>
            </fixed:field>
            <fixed:sequence name="shippingAddress">
              <fixed:field name="name" size="30"/>
              <fixed:field name="street1" size="100"/>
```

```
                <fixed:field name="street2" size="100"/>
                <fixed:field name="city" size="20"/>
                <fixed:field name="state" size="2"/>
                <fixed:field name="zip" size="5"/>
              </fixed:sequence>
</fixed:sequence>
          </fixed:body>
        </input>
        <output name="widgetOrderBill">
          <fixed:body>
<fixed:sequence name="widgetOrderConformation">
              <fixed:field name="amount" format="###"/>
              <fixed:field name="order_date"
              format="MM/DD/YYYY"/>
              <fixed:field name="type" size="2">
                <fixed:enumeration value="big"
                fixedValue="bg"/>
                <fixed:enumeration value="large"
                fixedValue="lg"/>
                <fixed:enumeration value="mungo"
                fixedValue="mg"/>
                <fixed:enumeration value="gargantuan"
                fixedValue="gg"/>
              </fixed:field>
              <fixed:field name="amtDue" format="####.##"/>
              <fixed:field name="orderNumber" size="20"/>
              <fixed:sequence name="shippingAddress">
                <fixed:field name="name" size="30"/>
                <fixed:field name="street1" size="100"/>
                <fixed:field name="street2" size="100"/>
                <fixed:field name="city" size="20"/>
                <fixed:field name="state" size="2"/>
                <fixed:field name="zip" size="5"/>
              </fixed:sequence>
</fixed:sequence>
          </fixed:body>
        </output>
    </operation>
  </binding>

  <service name="orderWidgetsService">
    <port name="widgetOrderPort"
    binding="tns:orderWidgetsBinding">
      <http:address location="http://localhost:8080"/>
</port>
</service>
</definitions>
```

Using Tagged Data

Artix has a binding that reads and writes messages where the data fields are delimited by specified characters.

The tagged data format supports applications that use self-describing, or delimited, messages to communicate. Artix can read tagged data and write it out in any supported data format. Similarly, Artix is capable of converting a message from any of its supported data formats into a self-describing or tagged data message.

You can enter the binding information using any text editor or XML editor.

Hand editing

To map a logical interface to a tagged data format do the following:

1. Add the proper namespace reference to the `definition` element of your contract. See Namespace.
2. Add a WSDL `binding` element to your contract to hold the tagged binding, give the binding a unique `name`, and specify the interface that represents the interface being bound.
3. Add a `tagged:binding` element as a child of the new `binding` element to identify this as a tagged binding and set the element's attributes to properly configure the binding.
4. For each operation defined in the bound interface, add a WSDL `operation` element to hold the binding information for the operation's messages.
5. For each operation added to the binding, add a `tagged:operation` child element to the `operation` element.
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. For each `input`, `output`, and `fault` element in the binding, add a `tagged:body` child element to define how the message parts are mapped into the concrete tagged data payload.

Namespace

The extensions used to describe tagged data bindings are defined in the namespace `http://schemas.iona.com/bindings/tagged`. Artix tools use the prefix tagged to represent the tagged data extensions. Add the following line to the `definitions` element of your contract:

```
xmlns:tagged="http://schemas.iona.com/bindings/tagged"
```

tagged:binding

`tagged:binding` specifies that the binding is for tagged data format messages. Its ten attributes are explained in Table 7.

Table 7. Attributes Used to Define a Tagged Binding

Attribute	Purpose
`selfDescribing`	Required attribute specifying if the message data on the wire includes the field names. Valid values are `true` or `false`. If this attribute is set to `false`, the setting for `fieldNameValueSeparator` is ignored.
`fieldSeparator`	Required attribute that specifies the delimiter the message uses to separate fields. Valid values include any character that is not a letter or a number.
`fieldNameValueSeparator`	Specifies the delimiter used to separate field names from field values in self-describing messages. Valid values include any character that is not a letter or a number.
`scopeType`	Specifies the scope identifier for complex messages. Supported values are `tab(\t)`, `curlybrace({`*`data`*`})`, and none. The default is `tab`.
`flattened`	Specifies if data structures are flattened when they are put on the wire. If `selfDescribing` is `false`, then this attribute is automatically set to `true`.
`messageStart`	Specifies a special token at the start of a message. It is used when messages require a special character at the start of the data sequence. Valid values include any character that is not a letter or a number.

Attribute	Purpose
`messageEnd`	Specifies a special token at the end of a message. Valid values include any character that is not a letter or a number.
`unscopedArrayElement`	Specifies if array elements need to be scoped as children of the array. If set to `true`, arrays take the form `echoArray{myArray=2;item=abc;item=def}`. If set to `false`, arrays take the form `echoArray{myArray=2;{0=abc;1=def;}}`. Default is `false`.
`ignoreUnknownElements`	Specifies if Artix ignores undefined elements in the message payload. Default is `false`.
`ignoreCase`	Specifies if Artix ignores the case with element names in the message payload. Default is `false`.

The settings for the attributes on these elements become the default settings for all the messages being mapped to the current binding.

tagged:operation

`tagged:operation` is a child element of the WSDL `operation` element and specifies that the operation's messages are being mapped to a tagged data format. It takes two optional attributes that are described in Table 8.

Table 8. Attributes for tagged:operation

Attribute	Purpose
`discriminator`	Specifies a discriminator to be used by the Artix runtime to identify the WSDL operation that will be invoked by the message receiver.
`discriminatorStyle`	Specifies how the Artix runtime will locate the discriminator as it processes the message. Supported values are `msgname`, `partlist`, `fieldvalue`, and `fieldname`.

tagged:body

`tagged:body` is a child element of the `input`, `output`, and `fault` messages being mapped to a tagged data format. It specifies that the message body is mapped to tagged data on the wire and describes the exact mapping for the message's parts.

`tagged:body` will have one or more of the following child elements:

- `tagged:field`
- `tagged:enumeration`
- `tagged:sequence`
- `tagged:choice`

They describe the detailed mapping of the message to the tagged data to be sent on the wire.

tagged:field

`tagged:field` is used to map simple types and enumerations to a tagged data format. Its two attributes are described in Table 9.

Table 9. Attributes for tagged:field

Attribute	Purpose
`name`	A required attribute that must correspond to the name of the logical message `part` that is being mapped to the tagged data field.
`alias`	An optional attribute specifying an alias for the field that can be used to identify it on the wire.

When describing enumerated types `tagged:field` will have a number of `tagged:enumeration` child elements.

tagged:enumeration

`tagged:enumeration` is a child element of `tagged:field` and is used to map enumerated types to a tagged data format. It takes one required attribute, `value`, which corresponds to the enumeration value as specified in the logical description of the enumerated type.

For example, if you had an enumerated type, `flavorType`, with the values `FruityTooty`, `Rainbow`, `BerryBomb`, and `OrangeTango` the logical description of the type would be similar to Example 43.

Example 43. Ice Cream Enumeration

```
<xs:simpleType name="flavorType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="FruityTooty"/>
    <xs:enumeration value="Rainbow"/>
    <xs:enumeration value="BerryBomb"/>
    <xs:enumeration value="OrangeTango"/>
  </xs:restriction>
</xs:simpleType>
```

`flavorType` would be mapped to the tagged data format shown in Example 44.

Example 44. Tagged Data Ice Cream Mapping

```
<tagged:field name="flavor">
  <tagged:enumeration value="FruityTooty"/>
  <tagged:enumeration value="Rainbow"/>
  <tagged:enumeration value="BerryBomb"/>
  <tagged:enumeration value="OrangeTango"/>
</tagged:field>
```

tagged:sequence

`taggeded:sequence` maps arrays and sequences to a tagged data format.

Its three attributes are described in Table 10.

Table 10. Attributes for tagged:sequence

Attributes	Purpose
`name`	A required attribute that must correspond to the name of the logical message part that is being mapped to the tagged data sequence.
`alias`	An optional attribute specifying an alias for the sequence that can be used to identify it on the wire.
`occurs`	An optional attribute specifying the number of times the sequence appears. This attribute is used to map arrays.

A `tagged:sequence` can contain any number of `tagged:field`, `tagged:sequence`, or `tagged:choice` child elements to describe the data contained within the sequence being mapped. For example, a structure containing a name, a date, and an ID number would contain three `tagged:field` elements to fully describe the mapping of the data to the fixed record message. Example 45 shows an Artix contract fragment for such a mapping.

Example 45. Mapping a Sequence to a Tagged Data Format

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="taggedDataMappingsample"
targetNamespace="http://www.iona.com/taggedSer vice"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:fixed="http://schemas.iona.com/bindings/tagged"
   xmlns:tns="http://www.iona.com/taggedService"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/taggedService"
xmlns="ht tp://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="person">
     <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="date" type="xsd:string"/>
       <xsd:element name="ID" type="xsd:int"/>
     </xsd:sequence>
   </xsd:complexType>
...
</types>
<message name="taggedSequence">
  <part name="personPart" type="tns:person"/>
</message>
<portType name="taggedSequencePortType">
...
</portType>
<binding name="taggedSequenceBinding"
       type="tns:taggedSequencePortType">
  <tagged:binding selfDescribing="false" fieldSeparator="pipe"/>
...
   <tagged:sequence name="personPart">
     <tagged:field name="name"/>
     <tagged:field name="date"/>
     <tagged:field name="ID"/>
   </tagged:sequence>
...
</binding>
...
</definition>
```

tagged:choice

tagged:choice maps unions to a tagged data format. Its three attributes are described in Table 11.

Table 11. Attributes for tagged:choice

Attributes	Purpose
name	A required attribute that must correspond to the name of the logical message part that is being mapped to the tagged data union.

Attributes	Purpose
discriminatorName	Specifies the message part used as the discriminator for the union.
alias	An optional attribute specifying an alias for the union that can be used to identify it on the wire.

A tagged:choice may contain one or more tagged:case child elements to map the cases for the union to a tagged data format.

tagged:case

tagged:case is a child element of tagged:choice and describes the complete mapping of a union's individual cases to a tagged data format. It takes one required attribute, name, that corresponds to the name of the case element in the union's logical description.

tagged:case must contain one child element to describe the mapping of the case's data to a tagged data format. Valid child elements are tagged:field, tagged:sequence, and tagged:choice.

Example 46 shows an Artix contract fragment mapping a union to a tagged data format.

Example 46. Mapping a Union to a Tagged Data Format

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
  targetNamespace="http://www.iona.com/tagService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:fixed="http://schemas.iona.com/bindings/tagged"
  xmlns:tns="http://www.iona.com/tagService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/tagService"
        xmlns="http://www.w3.org/2001/XMLS chema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="unionStationType">
      <xsd:choice>
        <xsd:element name="train" type="xsd:string"/>
        <xsd:element name="bus"     type="xsd:int"/>
        <xsd:element name="cab"     type="xsd:int"/>
        <xsd:element name="subway" type="xsd:string"/>
      </xsd:choice>
   </xsd:complexType>
...
</types>
<message name="tagUnion">
    <part name="stationPart" type="tns:unionStationType"/>
</message>
<portType name="tagUnionPortType">
```

```
...
</portType>
<binding name="tagUnionBinding" type="tns:tagUnionPortType">
  <tagged:binding selfDescribing="false"
        fieldSeparator="comma"/>
...
   <tagged:choice name="stationPart"
         descriminatorName="disc">
   <tagged:case name="train">
   <tagged:field name="name"/>
      </tagged:case>
     <tagged:case name="bus">
       <tagged:field name="number"/>
     </tagged:case>
     <tagged:case name="cab">
       <tagged:field name="number"/>
     </tagged:case>
     <tagged:case name="subway">
       <tagged:field name="name"/>
     </tagged:case>
</tagged:choice>
...
</binding>
...
</definition>
```

Example

Example 47 shows an example of an Artix contract containing a tagged data format binding.

Example 47. Tagged Data Format Binding

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="widgetOrderForm.wsdl"
targetNamespace="http://widgetVendor.com/widgetOrderForm"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://widgetVendor.com/widgetOrderForm"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:taged="http://schames.iona.com/binings/tagged"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
  <schema
        targetNamespace="http://widgetVendor.com/types/widget
        Types" xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <xsd:simpleType name="widgetSize">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="big"/>
      <xsd:enumeration value="large"/>
      <xsd:enumeration value="mungo"/>
      <xsd:enumeration value="gargantuan"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="Address">
  <xsd:sequence>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:element name="street1" type="xsd:string"/>

  <xsd:element name="street2" type="xsd:string"/>
  <xsd:element name="city" type="xsd:string"/>
  <xsd:element name="state" type="xsd:string"/>
  <xsd:element name="zipCode" type="xsd:string"/>
  </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="widgetOrderInfo">
  <xsd:sequence>
  <xsd:element name="amount" type="xsd:int"/>
  <xsd:element name="order_date" type="xsd:string"/>
  <xsd:element name="type" type="xsd1:widgetSize"/>
  <xsd:element name="shippingAddress" type="xsd1:Address"/>
  </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="widgetOrderBillInfo">
  <xsd:sequence>
  <xsd:element name="amount" type="xsd:int"/>
  <xsd:element name="order_date" type="xsd:string"/>
  <xsd:element name="type" type="xsd1:widgetSize"/>
  <xsd:element name="amtDue" type="xsd:float"/>
  <xsd:element name="orderNumber" type="xsd:string"/>
  <xsd:element name="shippingAddress" type="xsd1:Address"/>
  </xsd:sequence>
  </xsd:complexType>
  </schema>
  </types>
  <message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
  </message>
  <message name="widgetOrderBill">
  <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
  </message>
  <portType name="orderWidgets">
  <operation name="placeWidgetOrder">
  <input message="tns:widgetOrder" name="order"/>
  <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
  </portType>
  <binding name="orderWidgetsBinding"
        type="tns:orderWidgets">
  <tagged:binding selfDescribing="false"
        fieldSeparator="pipe"/>
  <operation name="placeWidgetOrder">
  <tagged:operation discriminator="widgetDisc"/>
  <input name="widgetOrder">
  <tagged:body>
  <tagged:sequence name="widgetOrderForm">
  <tagged:field name="amount"/>
  <tagged:field name="order_date"/>
  <tagged:field name="type" >
<tagged:enumeration value="big"/>
             <tagged:enumeration value="large"/>
             <tagged:enumeration value="mungo"/>
             <tagged:enumeration value="gargantuan"/>
           </tagged:field>
           <tagged:sequence name="shippingAddress">
             <tagged:field name="name"/>
             <tagged:field name="street1"/>
             <tagged:field name="street2"/>
             <tagged:field name="city"/>
             <tagged:field name="state"/>
             <tagged:field name="zip"/>
           </tagged:sequence>
</tagged:sequence>
        </tagged:body>
      </input>
      <output name="widgetOrderBill">
        <tagged:body>
<tagged:sequence name="widgetOrderConformation">
           <tagged:field name="amount"/>
<tagged:field name="order_date"/>
           <tagged:field name="type">
             <tagged:enumeration value="big"/>
             <tagged:enumeration value="large"/>
             <tagged:enumeration value="mungo"/>
             <tagged:enumeration value="gargantuan"/>
           </tagged:field>
           <tagged:field name="amtDue"/>
           <tagged:field name="orderNumber"/>
           <tagged:sequence name="shippingAddress">
             <tagged:field name="name"/>
             <tagged:field name="street1"/>
             <tagged:field name="street2"/>
             <tagged:field name="city"/>
             <tagged:field name="state"/>
             <tagged:field name="zip"/>
           </tagged:sequence>
</tagged:sequence>
        </tagged:body>
      </output>
   </operation>
</binding>
  <service name="orderWidgetsService">
<port name="widgetOrderPort"
binding="tns:orderWidgetsBinding">
<http:address location="http://localhost:8080"/>
</port>
</service>
</definitions>
```

Using the Pass Through Binding

The pass through binding allows you to send untyped message buffers through the run time as binary blobs. The runtime will make no attempts to interpret the data in the message.

The pass through binding is a simple binding for sending blobs of data across the wire. The binding makes no attempts to interpret the data and just writes the message to the wire as a string. It is ideal for dealing with data that uses custom encodings that cannot be properly described using one of the other Artix ESB bindings.

When sending data, an application can just simply stuff the binary data into a string buffer. The pass through binding will simply dump the data to the wire.

When reading data, the pass through binding simply reads messages into a string buffer. The string buffer is passed to the application layer. The binding does not make any effort to interpret the data or change it in anyway. The string buffer can be unpacked into an exact copy of the message sent across the wire.

Limitations

Because the pass through binding makes no effort to interpret the message contents and only interacts with string buffers it imposes some limitation on how you define your service's interface:

- Input messages can only have a single string part.
- Output messages can only have a single string part.
- On the server-side, requests are always handled by the first operation defined in the WSDL's `portType` element.

Namespace

The extensions used to describe the pass though binding are defined in the namespace http://schemas.iona.com/bindings/passthru. Add the following line to your contracts:

```
xmlns:passthru="http://schemas.iona.com/bindings/passthru"
```

Describing the binding

You describe a pass through binding by adding a single `passthru:binding` child to the WSDL `binding` element.

Example 48. Pass Through Binding Example

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.iona.com/bus/tests"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.iona.com/transports/http"
  xmlns:passthru="http://schemas.iona.com/bindings/passthru
  " targetNamespace="http://www.iona.com/bus/tests"
  name="PassthruService">
  <message name="returnString" />
  <message name="returnStringResponse">
    <part name="return" type="xsd:string"/>
  </message>
  <portType name="PassthruPortType">
    <operation name="returnString">
      <input name="returnString"
      message="tns:returnString"/>
      <output name="returnStringResponse"
      message="tns:returnStringResponse"/>
    </operation>
  </portType>
  <binding name="PassthruPortBinding"
  type="tns:PassthruPortType">
    <passthru:binding combineParts="true" />
  </binding>
  ...
</definitions>
```

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 the tools that 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 for C++ supports. Artix ESB C++ Runtime supports the following transports:

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

Using HTTP

HTTP is the standard TCP/IP-based protocol used for client-server communications on the World Wide Web. The main function of HTTP is to establish a connection between a web browser (client) and a web server for the purposes of exchanging files and possibly other information on the Web.

Adding an HTTP Endpoint to a Contract

Artix provides three ways of specifying an HTTP endpoint's address depending on the payload format you are using. SOAP 1.1 has a standardized `soap:address` element. SOAP 1.2 uses the `wsoap12:address` element. All other payload formats use Artix's `http:address` element.

As well as the standard `soap:address` element or `http:address` element, Artix provides a number of HTTP extensions. The Artix extensions allow you to specify a number of the HTTP port's configuration values in the contract.

SOAP 1.1

When you are sending SOAP 1.1 messages over HTTP you must use the `soap:address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL.

Example 49 shows a `port` element used to send SOAP 1.1 messages over HTTP.

Example 49. SOAP 1.1 Port Element

```
<service name="artieSOAP11Service">
   <port binding="artieSOAPBinding" name="artieSOAPPort">
<soap:address location="http://artie.com/index.xml">
  </port>
</service>
```

SOAP 1.2

When you are sending SOAP 1.2 messages over HTTP you must use the `wsoap12:address` element to specify the endpoint's address. It has one attribute, `location`, that specifies the endpoint's address as a URL.

Example 49 shows a `port` element used to send SOAP 1.2 messages over HTTP.

Example 50. SOAP 1.2 Port Element

```
<service name="artieSOAP12Service">
  <port binding="artieSOAPBinding" name="artieSOAPPort">
    <wsoap12:address location="http://artie.com/index.xml">
 </port>
</service>
```

Other payloads

When your messages are mapped to any payload format other than SOAP, such as fixed, you must use Artix's `http:address` element to specify the endpoint's address. Like the `soap:address` element, it has one attribute, `location`, that specifies the endpoint's address as a URL.

Using the command line tool

To use **wsdltoservice** to add an HTTP endpoint, use the following options.

```
wsdltoservice {-transport soap/http} [-e service]
[-t port] [-b binding] [-a address] [-hssdt
serverSendTimeout] [-hscvt serverReceiveTimeout]
[-hstrc trustedRootCertificates] [-hsuss
useSecureSockets] [-hsct contentType] [-hscc
serverCacheControl] [-hsscse
supressClientSendErrors] [-hsscre
supressClientReceiveErrors] [-hshka honorKeepAlive]
[-hsmps serverMultiplexPoolSize] [-hsrurl
redirectURL] [-hscl contentLocation] [-hsce
contentEncoding] [-hsst serverType] [-hssc
serverCertificate] [-hsscc serverCertificateChain]
[-hsspk serverPrivateKey] [-hsspkp
serverPrivateKeyPassword] [-hcst clientSendTimeout]
[-hccvt clientReceiveTimeout] [-hctrc
trustedRootCertificates] [-hcuss useSecureSockets]
[-hcct contentType] [-hccc clientCacheControl]
[-hcar autoRedirect] [-hcun userName] [-hcp
password] [-hcat clientAuthorizationType] [-hca
clientAuthorization] [-hca accept] [-hcal
acceptLanguage] [-hcae acceptEncoding] [-hch host]
[-hccn clientConnection] [-hcck cookie] [-hcbt
browserType] [-hcr referer] [-hcps proxyServer]
[-hcpun proxyUserName] [-hcpp proxyPassword] [-hcpat
proxyAuthorizationType] [-hcpa proxyAuthorization]
[-hccce clientCertificate] [-hcccc
clientCertificateChain] [-hcpk clientPrivateKey]
[-hcpkp clientPrivateKeyPassword] [-o file] [-d
dir] [-L file] [[-quiet] | [-verbose]] [-h] [-v]
wsdlurl
```

The `-transport soap/http` flag specifies that the tool is to generate an HTTP service. The other options are as follows.

Table 22. Options for Adding an HTTP Endpoint

Option	Description
`-transport soap/http`	If the payload being sent over the wire is SOAP, use `-transport soap`. For all other payloads use `-transport http`.
`-e` *`service`*	Specifies the name of the generated `service` element.
`-t` *`port`*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *`binding`*	Specifies the name of the binding for which the service is generated.
`-a` *`address`*	Specifies the value used in the `address` element of the port.
`-hssdt` *`serverSendTimeout`*	Specifies the number of milliseconds that the server can continue to try to send a response to the client before the connection is timed-out.
`-hscvt` *`serverReceiveTimeout`*	Specifies the number of milliseconds that the server can continue to try to receive a request from the client before the connection is timed-out.
`-hstrc` *`trustedRootCertificates`*	Specifies the full path to the X509 certificate for the certificate authority.
`-hsuss` *`useSecureSockets`*	Specifies if the server uses secure sockets. Valid values are `true` or `false`.
`-hsct` *`contentType`*	Specifies the media type of the information being sent in a server response.
`-hscc` *`serverCacheControl`*	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
`-hsscse` *`supressClientSendErrors`*	Specifies whether exceptions are thrown when an error is encountered on receiving a client request. Valid values are `true` or `false`.
`-hsscre` *`supressClientReceiveErrors`*	Specifies whether exceptions are thrown when an error is encountered on sending a response to a client. Valid values are `true` or `false`.
`-hshka` *`honorKeepAlive`*	Specifies if the server honors client keep-alive requests. Valid values are `true` or `false`.
`-hsrurl` *`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.

Option	Description
-hscl *contentLocation*	Specifies the URL where the resource being sent in a server response is located.
-hsce *contentEncoding*	Specifies what additional content codings have been applied to the information being sent by the server, and what decoding mechanisms the client therefore needs to retrieve the information.
-hsst *serverType*	Specifies what type of server is sending the response to the client.
-hssc *serverCertificate*	Specifies the full path to the X509 certificate issued by the certificate authority for the server.
-hsscc *serverCertificateChain*	Specifies the full path to the file that contains all the certificates in the chain.
-hsspk *serverPrivateKey*	Specifies the full path to the private key that corresponds to the X509 certificate specified by *serverCertificate*.
-hsspkp *serverPrivateKeyPassword*	Specifies a password that is used to decrypt the private key.
-hcst *clientSendTimeout*	Specifies the number of milliseconds that the client can continue to try to send a request to the server before the connection is timed-out.
-hccvt *clientReceiveTimeout*	Specifies the number of milliseconds that the client can continue to try to receive a response from the server before the connection is timed-out.
-hctrc *trustedRootCertificates*	Specifies the full path to the X509 certificate for the certificate authority.
-hcuss *ueSecureSockets*	Specifies if the client uses secure sockets. Valid values are `true` or `false`.
-hcct *contentType*	Specifies the media type of the data being sent in the body of the client request.
-hccc *clientCacheControl*	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
-hcar *autoRedirect*	Specifies if the server should automatically redirect client requests.
-hcun *userName*	Specifies the username the client uses to register with servers.
-hcp *password*	Specifies the password the client uses to register with servers.

Option	Description
-hcat *clientAuthorizationType*	Specifies the authorization mechanisms the client uses when contacting servers.
-hca *clientAuthorization*	Specifies the authorization credentials used to perform the authorization.
-hca *accept*	Specifies what media types the client is prepared to handle.
-hcal *acceptLanguage*	Specifies what language the client prefers for the purposes of receiving a response
-hcae *acceptEncoding*	Specifies what content codings the client is prepared to handle.
-hch *host*	Specifies the internet host and port number of the resource on which the client request is being invoked.
-hccn *clientConnection*	Specifies if the client will open a new connection for each request or if it will keep the original one open. Valid values are `close` and `Keep-Alive`.
-hcck *cookie*	Specifies a static cookie to be sent to the server.
-hcbt *browserType*	Specifies information about the browser from which the client request originates.
-hcr *referer*	Specifies the value for the client’s referring entity.
-hcps *proxyServer*	Specifies the URL of the proxy server, if one exists along the message path.
-hcpun *proxyUserName*	Specifies the username that the client uses to be authorized by proxy servers.
-hcpp *proxyPassword*	Specifies the password that the client uses to be authorized by proxy servers.
-hcpat *proxyAuthorizationType*	Specifies the authorization mechanism the client uses with proxy servers.
-hcpa *proxyAuthorization*	Specifies the actual data that the proxy server should use to authenticate the client.
-hccce *clientCertificate*	Specifies the full path to the X509 certificate issued by the certificate authority for the client.
-hcccc *clientCertificateChain*	Specifies the full path to the file that contains all the certificates in the chain.
-hcpk *clientPrivateKey*	Specifies the full path to the private key that corresponds to the X509 certificate specified by *clientCertificate*.

Option	Description
`-hcpkp` *`clientPrivateKeyPassword`*	Specifies a password that is used to decrypt the private key.
`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d dir`	Specifies the output directory for the generated contract.
`-L file`	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.
`-h`	Displays the tool's usage statement.
`-v`	Displays the tool's version.

For more information about the specific attributes and their values see the ***Artix WSDL Extension Reference***.

Example

Example 51 shows the namespace entries you need to add to the `definitions` element of your contract to use the HTTP extensions.

Example 51. Artix HTTP Extension Namespaces

```
<definitions
  ...
  xmlns:http="http://schemas.iona.com/transports/http"
  ... >
```

Example 52 shows a `port` element for an endpoint that sends fixed data over HTTP.

Example 52. Generic HTTP Port

```
<service name="artieFixedService">
  <port binding="artieFixedBinding" name="artieFixedPort">
<http:address location="http://artie.com/index.xml">
  </port>
</service>
```

Configuring an HTTP Endpoint

In addition to the `http:address` element or `soap:address` element used to specify the URL of an HTTP endpoint, Artix uses two other elements to define a number of other properties for HTTP endpoints: `http-conf:client` and `http-conf:server`.

The `http-conf:client` element specifies properties used to configure an HTTP client-side endpoint. The `http-conf:server` element specifies properties used to configure an HTTP server-side endpoint. The properties are specified as attributes to the elements. While the elements share many attributes there are differences.

To use the HTTP configuration elements, you need to include the following entry in your contract's `definition` element:

```
xmlns:http-conf=
"http://schemas.iona.com/transports/http/configuration"
```

For a complete discussion of the specific attributes and their values see the ***Artix WSDL Extension Reference***.

Specifying Send and Receive Timeout Limits

The most common values that needs to be configured for an HTTP endpoint are the ones controlling how long the endpoint will spend sending a receiving messages before issuing a timeout exception. Both client endpoints and server endpoints have two attributes that control their timeout behaviors: `SendTimeout` and `RecieveTimeout`.

Send timeout

The timeout limit for attempting to send a message is specified, for both the client-side and server-side, using the `SendTimeout` attribute. The timeout limit specifies the number of milliseconds an endpoint will spend attempting to transmit a message. It has a default setting of 30000 milliseconds.

This value may need to be adjusted if you are transmitting large messages as they take longer to send. Other factors that may affect the amount of time needed to transmit messages over HTTP are the speed of the network, distance between the endpoints, and the amount of traffic on the network. For example, if you were transmitting high-resolution photographs across the Atlantic, you may need to adjust the value of the `SendTimeout` attribute to 1200000 as shown in Example 53.

Example 53. Setting the SendTimeout Attribute

```
<port ...>
  <soap:address ... />
  <http-conf:client SendTimeout="120000" />
</port>
```

Receive timeout

The `ReceiveTimeout` attribute specifies the amount of time an endpoint spends between when it initially receives the beginning of a message and the when it receives the last piece of data in the message. For example, if a client using the default settings sends a response to a service that takes 90 seconds to process the response, the client will not timeout. However, if it takes the client 45 seconds to read the response from the network, it will timeout.

The causes for long read times are similar to the reasons for long send times. Large messages, heavy network traffic, and large physical distances can all have an impact on the amount of time it takes an HTTP endpoint to receive a message. For example, if you are transmitting map data to a remote research facility, you may want to specify a value of 600000 for the `ReceiveTimeout` attribute of the remote endpoint as shown in Example 54.

Example 54. Setting the ReceiveTimeout Attribute

```
<port ...>
  <soap:address ... />
  <http-conf:server ReceiveTimeout="600000" />
</port>
```

Specifying a Username and a Password

Username/password authentication is a common way of requiring clients to identify themselves. By requiring a client to provide a username and a password, a server can keep a record of who is accessing it and determine if they are authorized to access the functionality requested. For example, many Wiki applications and blogging applications require a username and password before allowing content to be edited.

In Artix, the username and password presented by an HTTP endpoint are specified using the following attributes of the `http-conf:client` element:

- `UserName`
- `Password`

Be aware that these values will be visible to anyone that has access to the endpoint's contract. Using this style of authentication does not provide a high level of security. For information on using stronger security measures with Artix see the ***Artix Security Guide.***

Setting a username

You set a username using the `http-conf:client` element's `UserName` attribute. The value you specify is used to populate the username field in the HTTP header of all messages sent from the endpoint. Setting this attribute is optional. If no value is specified, Artix does not populate the username field of the HTTP header with a default value.

Setting a password

You set a password using the `http-conf:client` element's `Password` attribute. The value you specify is used to populate the password field in the HTTP header of all messages sent from the endpoint. It is an entirely optional attribute. If no value is specified, Artix does not populate the password field of the HTTP header with a default value.

Relationship between the attributes

The `UserName` attribute and the `Password` attribute are independent of each other. Although most applications that require a username also require a password, it is not mandatory that this pattern is followed. An application may just require a username for identification, or it may just use a password to provide a level of exclusivity.

Similarly, Artix does not require that the two attributes be used together. If an endpoint only needs to provide a password, you can provide a value for the `Password` attribute without providing a value for the `UserName` attribute. Example 55 shows an HTTP endpoint definition that specifies just a username.

Example 55. Specifying Just a Username

```
<port ...>
  <http:address ... />
  <http-conf:client UserName="Joe" />
</port>
```

The attributes and other security features

Specifying a username and password in an endpoint's contract does not affect the use of other Artix security features. You are not forced to use HTTPS when using a username or password. Similarly, you are not stopped from implementing your endpoint using WS-Security headers. For more details on using Artix's security features see the ***Artix Security Guide***.

Configuring Keep-Alive Behavior

The default behavior of Artix endpoints is to open a connection and keep it open for as long as the client requires. However, it is not always desirable to keep a connection open over multiple requests. This can present a security problem. Artix endpoints can, therefore, be configured to close connections after each request/response cycle.

Making keep-alive requests

HTTP client endpoints are configured to make keep-alive requests using the `http-conf:client` element's `Connection` attribute. This attribute has two values: `close` and `Keep-Alive`.

`Keep-Alive` is the default. It specifies that the client endpoint wishes to keep its connections open for future requests. The client will request that the server keep the connection open. If the server does honor the request, the connection remains open until one of the endpoints dies. If the server does not honor the request, the client must open a new connection for each request.

`close` specifies that the client endpoint does not wish to keep its connections open for future requests. The client will always open a new connection for each request.

Example 56 shows a `port` element that defines an HTTP client endpoint that does not want to reuse connections.

Example 56. Specifying that the HTTP Connection is Closed

```
<port ...>
  <soap:address location="http://localhost:8080" />
  <http-conf:client Connection="close" />
</port>
```

Honoring keep-alive requests

HTTP server endpoints are not required to honor keep-alive requests. The default behavior of Artix HTTP server endpoints is the accept keep-alive requests. You can change this behavior using the `http-conf:server` element's `HonorKeepAlive` attribute. It has two values: `false` and `true`.

`true` is the default. It specifies that the server endpoint will honor all keep-alive requests. If a client connects to the server endpoint using at least HTTP 1.1 and requests that the connection is kept alive, the server endpoint is left open. The client can continue to make requests over the original connection.

`false` specifies that the server endpoint rejects all keep-alive requests. Once the endpoint responds to a request it closes the connection used for the request/response sequence.

Example 57 shows a `port` element that defines an HTTP server endpoint that rejects keep-alive requests.

Example 57. Rejecting Keep-Alive Requests

```
<port ...>
  <soap:address location="http://localhost:8080" />
  <http-conf:server HonorKeepAlive="false" />
</port>
```

Specifying Cache Control Directives

A common method to reduce latency and control network traffic on the Web is to use caches that sit between server endpoints and client endpoints. These caches monitor the interactions between the endpoints. They store responses to requests as they are passed from a server endpoint to a client endpoint.

When a cache sees a request that it recognizes, it will check its stored responses. If a match is found, the cache will respond to the request on behalf of the server endpoint. The server endpoint will never know the request was made and the client endpoint will never know that it is getting a cached response.

While this optimizes the transaction time, it does pose a few possible problems:

- If a server endpoint collects usage statistics, it will not have accurate data.
- If the server endpoint frequently updates its data, the client endpoint may get a response that is out of date.

HTTP provides a mechanism for specifying cache behavior using the HTTP message header. You can configure these settings for your endpoints using the `CacheControl` attribute of both the `http-conf:server` element and the `http-conf:client` element.

Server endpoint settings

Server endpoints can tell caches how to handle the responses they issue. For example, a server endpoint can direct caches that its responses are stale after 10 seconds. These directives are only valid for the responses issued from a particular server endpoint.

Table 23 shows the valid values for `CacheControl` in `http-conf:server`.

Table 23. Settings for CacheControl on an HTTP Server Endpoint

Directive	Behavior
`no-cache`	Caches cannot use a particular response to satisfy subsequent client requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`public`	Any cache can store the response.
`private`	Public (*shared*) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`no-store`	Caches must not store any part of response or any part of the request that invoked it.
`no-transform`	Caches must not modify the media type or location of the content in a response between a server and a client.
`must-revalidate`	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.
`proxy-revalidate`	Means the same as `must-revalidate`, except that it can only be enforced on shared caches and is ignored by private unshared caches. If using this directive, the `public` cache directive must also be used.
`max-age`	Specifies the maximum age, in seconds, of a cached response before it is stale.
`s-maxage`	Means the same as `max-age`, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by `s-maxage` overrides the age specified by `max-age`. If using this directive, the `proxy-revalidate` directive must also be used.
`cache-extension`	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

Client endpoint settings

Client endpoints can tell caches what kinds of responses they will accept and how to handle the response they receive. For example, a client endpoint can direct caches not to store any responses that it receives. A client endpoint can also direct caches that it will only accept a cached response that is less than 5 seconds old.

Table 24 shows the valid settings for `CacheControl` in `http-conf:client`.

Table 24. Settings for CacheControl on HTTP Client Endpoint

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

Managing Cookies in Artix Clients

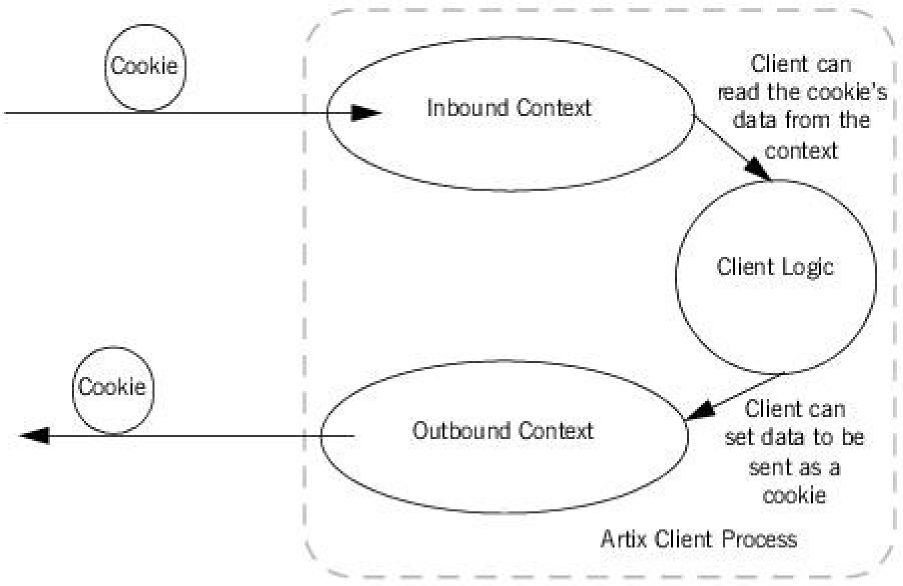
Artix can send and receive cookies. It can also be configured to pass along a static cookie with all outgoing requests. While Artix can send and receive cookies, it is up to the application to set dynamic cookies and ensure they are properly managed.

Sending static cookies

If you want your client to always attach a static cookie to its requests, you can specify this in the client's contract. The cookie is specified using the `cookie` attribute of the `http-conf:client` element.

How Artix processes cookies Artix handles cookies using its context mechanism. For an HTTP application there are two contexts. One context is for incoming messages and the other is for outgoing messages. Figure 1 shows how an Artix client manages cookies.

Figure 1. Artix Cookie Processing


Cookie
Inbound Context
Client can read the cookie's data from the context
Client Logic
Cookie
Outbound Context
Client can set data to be sent as a cookie
Artix Client Process

When a client makes a request it can save a cookie into its outbound context and it will be sent with all future requests. If a client receives a cookie from a service, that cookie is stored in the client's inbound context.

The received cookie does not have to be inspected. In order to inspect the contents of a received cookie, you will need to add the proper logic to your client using the Artix context APIs.

The received cookie is not automatically transferred to the out bound context. If you client needs to pass a received cookie along with future requests, you will need to add logic to your client so that it will transfer the received cookie from the client's inbound context to the outbound context.

More information

For information about setting cookies and using Artix contexts, see ***Developing Artix Applications in C++***.

Using the Java Messaging System

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

Defining a JMS Endpoint

Artix provides a transport plug-in that enables endpoints to use Java Messaging System (JMS) queues and topics. One large advantage of this is that Artix allows C++ applications to interact directly with Java applications over JMS.

Artix's JMS transport plug-in uses the Java Naming and Directory Interface (JNDI) to locate and obtain references to the JMS provider. Once Artix has established a connection to a JMS provider, Artix supports the passing of messages packaged as either a JMS `ObjectMessage` or a JMS `TextMessage`.

Message formatting

The JMS transport takes messages and packages them into either a JMS `ObjectMessage` or a `TextMessage`. When a message is packaged as an `ObjectMessage` the message's data, including any format-specific information, is serialized into a byte[] and placed into the JMS message body. When a message is packaged as a `TextMessage`, the message's data, including any format-specific information, is converted into a string and placed into the JMS message body.

When a message sent by Artix is received by a JMS application, the JMS application is responsible for understanding how to interpret the message and the formatting information. For example, if the Artix contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as a `TextMessage`, the JMS application will receive a string containing all of the SOAP envelope information. For a message encoded using the fixed binding, the message will contain no formatting information, simply a string of characters, numbers, and spaces.

Namespace

The WSDL extensions used to define a JMS endpoint are specified in the namespace http://celtix.objectweb.org/transports/jms. To use the JMS extensions you will need to add the line shown Example 58 to the `definitions` element of your contract.

Example 58. JMS Extension's Namespace

```
xmlns:jms="http://celtix.objectweb.org/transports/jms"
```

Basic Endpoint Configuration

JMS endpoints need to know certain basic information about how to establish a connection to the proper destination. This information is provided using the `jms:address` element and its child the `jms:JMSNamingProperty` element. The `jms:address` element's attributes specify the information needed to identify the JMS broker and the destination. The `jms:JMSNamingProperty` element specifies the Java properties used to connect to the JNDI service.

address element

The basic configuration for a JMS endpoint is done by using a `jms:address` element in your service's `port` element. The `jms:address` element uses the attributes described in Table 25 to configure the connection to the JMS broker.

Table 25. JMS Port Attributes

Attribute	Description
`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`jndiConnectionFactoryName`	Specifies the JNDI name of the JMS connection factory to use when connecting to the JMS destination.
`jmsDestinationName`	Specifies the JMS name of the destination to which requests are sent.
`jmsReplyDestinationName`	Specifies the JMS name of the destination where replies are sent. This attribute allows you to use a user defined destination for replies. For details see Using a named reply destination.
`jndiDestinationName`	Specifies the JNDI name of the destination to which requests are sent.
`jndiReplyDestinationName`	Specifies the JNDI name of the destination where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Using a named reply destination.
`connectionUserName`	Specifies the username to use when connecting to a JMS broker.
`connectionPassword`	Specifies the password to use when connecting to a JMS broker.

JMSNamingProperties element

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

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

- java.naming.factory.initial
- java.naming.provider.url
- java.naming.factory.object
- java.naming.factory.state
- java.naming.factory.url.pkgs
- java.naming.dns.url
- java.naming.authoritative
- java.naming.batchsize
- java.naming.referral
- java.naming.security.protocol
- java.naming.security.authentication
- java.naming.security.principal
- java.naming.security.credentials
- java.naming.language
- java.naming.applet

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

Using a named reply destination

By default Artix ESB for C++ endpoints using JMS create a temporary queue for the response queue. You can change this behavior by setting either the `jmsReplyDestinationName` attribute or the `jndiReplyDestinationName` attribute in the endpoint's contract. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the `ReplyTo` field of all outgoing requests. An service endpoint will use the value of the `jndiReplyDestinationName` attribute as the location for placing replies if there is no destination specified in the request's `ReplyTo` field.

Examples

Example 59 shows an example of a JMS port specification that uses dynamic queues.

Example 59. Artix JMS Port with DynamicQueues

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

Example 60 shows an example of a JMS port specification that does not use dynamic queues.

Example 60. Artix JMS Port with Non-dynamic Queues

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
         jmsDestinationName="sonic:jms/queue/RequestQJmsDestination"
         destinationStyle="queue">
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                 value="org.activemq.jndi.ActiveMQInitialContextFactory"
                 />
      <jms:JMSNamingProperty name="java.naming.provider.url"
      value="tcp://localhost:61616"
 />
      <jms:JMSNamingProperty name="queue.MyQueue"
      value="example.MyQueue" />
    </jms:address>
  </port>
</service>
```

Alternate InitialContextFactory settings for using SonicMQ

If you are using Sonic MQ, you must use an alternative method of specifying the `InitialContextFactory` value. Specify a colon-separated list of package prefixes to force the

JNDI service to instantiate a context factory with the class name `com.iona.jbus.jms.naming.sonic.sonicURLContextFactory` to perform lookups. This is shown in Example 61.

Example 61. JMS Port with Alternate InitialContextFactory Specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address
             jndiConnectionFactoryName="sonic:jms/queue/connectionFactory"
             jndiDestinationName="jms/queue/helloWorldQueue">
      <jms:JMSNamingProperty name="java.naming.factory.initial"
           value="com.iona.jbus.jms.naming.sonic.sonicURLContextFactory"
 />
      <jms:JMSNamingProperty name="java.naming.provider.url"
      value="tcp://localhost:61616"
 />
    </jms:address>
  </port>
</service>
```

Using the contract in Example 76 on page 169, Artix would use the URL `sonic:jms/queue/helloWorldQueue` to get a reference to the desired queue. Artix would be handed a reference to a queue named `helloWorldQueue` if the JMS broker has such a queue.

Client Endpoint Configuration

The client endpoint's behaviors are configured using the `jms:client` element. The `jms:client` element is a child of the WSDL `port` element and has one attribute:

Table 26. Attributes for Configuring a JMS Client Endpoint

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

This element is optional. The default behavior of a JMS client endpoint is to send text messages.

Server Endpoint Configuration

JMS server endpoints have a number of behaviors that are configurable in the contract. These include if the server uses durable subscriptions, if the server uses local JMS transactions, and the message selectors used by the endpoint.

server element

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

Table 27. Attributes for Configuring a JMS Server Endpoint

Attribute	Description
`useMessageIDAsCorrealationID`	Specifies whether JMS will use the message ID to correlate messages. The default is `false`.
`durableSubscriberName`	Specifies the name used to register a durable subscription. See Setting up durable subscriptions.
`messageSelector`	Specifies the string value of a message selector to use. See Using message selectors bookmark298.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false`. See Using reliable messaging.

The `jms:server` element and all of its attributes are optional.

Setting up durable subscriptions

If you want to configure your server to use durable subscriptions, you can set the optional `durableSubscriberName` attribute. The value of the attribute is the name used to register the durable subscription.

Using message selectors

If you want to configure your server to use a JMS message selector, you can set the optional `messageSelector` attribute. The value of the attribute is the string value of the selector. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.

Using reliable messaging

If you want your server to use the local JMS broker's transaction capabilities, you can set the optional `transactional` attribute to `true`.

When the `transactional` attribute is set, an Artix server's JMS transport layer will begin a transaction when it pulls a request from the queue. The server will then process the request and send the response back to the JMS transport layer. Once the JMS transport layer has successfully placed the response on the response queue, the transport layer will commit the transaction. So, if the Artix server crashes while processing a request or the transport layer is unable to send the response, the JMS broker will hold the request in the queue until it is successfully processed.

In cases where Artix is acting as a router between JMS and another transport, setting the `transactional` attribute will ensure that the message is delivered to the second server. The JMS portion of the router will not commit the message until the message has been successfully consumed by the outbound transport layer. If an exception is thrown during the consumption of the message, the JMS transport will rollback the message, pull it from the queue again, and attempt to resend it.

Using the Command Line Tool

To use **wsdltoservice** to add a JMS endpoint use the tool with the following options:

wsdltoservice {-transport jms} [-e *service*] [-t *port*] [-b *binding*] [-o *file*] [-d *dir*] [-jnp *propName*:*propVal*...] [-jds [queuetopic]] [-jnf *connectionFactoryName*] [-jdn *destinationName*] [-jrdn *replyDesinationName*] [-jcun *username*] [-jcp *password*] [-jmt [textbinar]] [-jms *messageSelector*] [-jumi [truefalse]] [-jtr [truefalse]] [-jdsn *durableSubscriber*] [-L *file*] [[-quiet] | [-verbose]] [-h] [-v] *wsdlurl*

The `-transport jms` flag specifies that the tool is to generate a JMS endpoint. The other options are as follows:

Table 28. Command Line Options for Creating a JMS Endpoint

Option	Description
`-e` *`service`*	Specifies the name of the generated `service` element.
`-t` *`port`*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *`binding`*	Specifies the name of the binding for which the service is generated.
`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *`dir`*	Specifies the output directory for the generated contract.
`-jnp` *`propName`*`:`*`propVal`*	Specifies any optional Java properties to use in connecting to the JNDI provider. This information is used to populate a `JMSNamingProperty` element.

Option	Description
	You can use this flag multiple times.
`-jds (queue/topic)`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`-jfn` *`connectionFactoryName`*	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`-jdn` *`destinationName`*	Specifies the JNDI name of the JMS destination to which Artix connects.
`-jrdn` *`replyDestinationName`*	Specifies the JNDI name of the JMS destination used for replies.
`-jcun` *`username`*	Specifies the username used to connect to the JMS broker.
`-jcp` *`password`*	Specifies the password used to connect to the JMS broker.
`-jmt (text/binary)`	Specifies how the message data will be packaged as a JMS message.
`-jms` *`messageSelector`*	Specifies a message selector to use when pulling messages from the JMS destination.
`-jumi (true/false)`	Specifies if the JMS message id should be used as the correlation id.
`-jtr (true/false)`	Specifies if the services uses local JMS transactions when processing requests.
`-jdsn` *`durableSubscriber`*	Specifies the name of the durable subscription to use.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT PRODUCT DIR\etc\license.txt`.
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.
`-h`	Displays the tool’s usage statement.
`-v`	Displays the tool’s version.

For more information about the specific attributes and their values see the ***Artix WSDL Extension Reference***.

Migrating to the 4.x JMS WSDL Extensions

The WSDL extensions used to configure a JMS endpoint were modified in the 4.0 release of Artix. This update makes Artix JMS endpoint definitions compatible with Celtix JMS endpoints. To make the transition as smooth as possible, Artix includes an XSLT script that can be used to automatically migrate an old JMS endpoint definition to a new JMS endpoint definition.

XSLT script

The XSLT script used to migrate old JMS endpoint definitions to 4.x JMS endpoints is called `oldjmswsdl_to_newjmswsdl.xsl` and it is located in

IntallDir/Artix/*Version*/etc/xslt/utilities/jms. It will take any Artix contract containing a pre-4.x Artix JMS endpoint definition as input and output an equivalent Artix contract containing a 4.x Artix JMS endpoint.

Using the script with Artix

You can use Artix's XSLT processor to convert your JMS endpoints. To do so you run the Artix **xslttransform** command line tool using the options shown in Example 62.

Example 62. Running the Transformer with the JMS Migration Script

```
xslttransform -XSL oldjmswsdl_to_newjmswsdl.xsl
-IN oldWsdl.wsdl
-OUT newWsdl.wsdl
```

The XSLT processor will read the contract in `oldWsdl.wsdl`, transform the old JMS endpoint to a new JMS endpoint, and save the resulting contract in `newWsdl.wsdl`.

Using ActiveMQ as Your JMS Provider

Artix installs ActiveMQ, an open source JMS implementation, for you to use as a possible messaging system. All of the Artix JMS demos are configured to use ActiveMQ, so to run the demos you must start the ActiveMQ broker.

Setting the CLASSPATH

When you set your Artix environment using the **artix_env** script, the ActiveMQ jars are automatically added to your `CLASSPATH`.

If you do not want to set the Artix environment before starting ActiveMQ you need to add *InstallDir*/lib/activemq/activemq/3.2.1/activemq-rt.jar to your CLASSPATH.

Starting the broker

To start the ActiveMQ JMS broker run the following command:

```
InstallDir/Artix/Version/bin/start_jms_broker
```

Stopping the broker

To shutdown the ActiveMQ JMS broker run the following command:

```
InstallDir/Artix/Version/bin/ jmsbrokerinteract -sd
```

Security

By default, ActiveMQ’s security features are turned off. To turn on ActiveMQ’s security features see the ActiveMQ documentation.

More information

For more information on using ActiveMQ see the project’s homepage at http://activemq.org.

Using IIOP

Using IIOP to send non-CORBA formats allows you to take advantages of CORBA services and QoS without using CORBA applications.

Artix allows you to use IIOP as a generic transport for sending data using any of the payload formats. When using IIOP as a generic transport, you define your endpoint's address using an`iiop:address` element. The benefit of using the generic IIOP transport is that it allows you to use CORBA services without requiring your applications to be CORBA applications. For example, you could use an IIOP tunnel to send fixed format messages to an endpoint whose address is published in a CORBA naming service.

Namespace

The namespace under which the IIOP extensions are defined is `http://schemas.iona.com/bindings/iiop_tunnel`. If you are going to add an IIOP port by hand you will need to add this to your contract's `definition` element.

IIOP address specification

The IOR, or address, of the IIOP port is specified using the `iiop:address` element. You have four options for specifying IORs in Artix contracts:

- Specify the object's IOR directly in the contract, using the stringified IOR format:

```
IOR:22342....
```

- Specify a file location for the IOR, using the following syntax:

```
file:///file_name
```

NOTE: The file specification requires three backslashes (///).

- Specify that the IOR is published to a CORBA name service, by entering the object's name using the `corbaname` format:

```
corbaname:rir/NameService#object_name
```

For more information on using the name service with Artix see ***Artix for CORBA.***

- Specify the IOR using `corbaloc`, by specifying the port at which the service exposes itself, using the `corbaloc` syntax.

```
corbaloc:iiop:host:port/service_name
```

When using corbaloc, you must be sure to configure your service to start up on the specified host and port.

Specifying type of payload encoding

The IIOP transport can perform codeset negotiation on the encoded messages passed through it if your CORBA system supports it. By default, this feature is disabled so that the agents sending the message maintain complete control over codeset conversion. If you wish to enable automatic codeset negotiation use the following element:

```
<iiop:payload type="string"/>
```

Specifying POA policies

Using the optional `iiop:policy` element, you can describe the POA polices Artix will use when creating the IIOP endpoint. These policies include:

- The POA name
- Persistence
- The system ID assigned to the POA

Setting these policies lets you exploit some of the enterprise features of Micro Focus Orbix 6.x, such as load balancing and fault tolerance, when deploying an Artix endpoints using the IIOP transport. For information on using these advanced CORBA features, see the Orbix documentation.

POA name

Artix POAs are created with the default name of `WS_ORB`. To specify a name for the POA that Artix creates for an IIOP endpoint, you use the following:

```
<iiop:policy poaname="poa_name"/>
```

The POA name is used for setting certain policies, such as direct persistence and well-known port numbers in the CORBA configuration.

Persistence

By default Artix POAs have a persistence policy of `false`. To set the POA's persistence policy to `true`, use the following:

```
<iiop:policy persistent="true"/>
```

ID Assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by Artix. To specify that the

IIOP endpoint's POA should use a user-assigned ID, use the following:

```
<corba:policy serviceid="POAid"/>
```

This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Using the command line tool

To use **wsdltoservice** to add an IIOP endpoint use the tool with the following options.

`wsdltoservice` {-transport iiop} [-e *service*] [-t *port*] [-b *binding*] [-a *address*] [-poa *poaName*] [-sid *serviceId*] [-pst *persists*] [-paytype *payload*] [-o *file*] [-d *dir*] [-L *file*] [[-quiet] | [-verbose]] [-h] [-v] *wsdlurl*

The `-transport iiop` flag specifies that the tool is to generate an IIOP endpoint. The other options are as follows.

`-e` *service*	Specifies the name of the generated `service` element.
`-t` *port*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *binding*	Specifies the name of the binding for which the endpoint is generated.
`-a` *address*	Specifies the value used in the generated `iiop:address` elements.
`-poa` *poaName*	Specifies the value of the POA name policy.
`-sid` *serviceId*	Specifies the value of the ID assignment policy.
`-pst` *persists*	Specifies the value of the persistence policy. Valid values are `true` and `false`.
`-paytype` *payload*	Specifies the type of data being sent in the message payloads. Valid values are `string`, `octets`, `imsraw`, `imsraw_binary`, `cicsraw`, and `cicsraw_binary`.
`-o` *file*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *dir*	Specifies the output directory for the generated contract.
`-L` *file*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.
`-h`	Displays the tool's usage statement.
`-v`	Displays the tool's version.

For more information about the specific attributes and their values see the ***Artix WSDL Extension Reference***.

Example

For example, an IIOP endpoint definition for the personalInfoLookup binding would look similar to Example 63:

Example 63. CORBA personalInfoLookup Port

```
<service name="personalInfoLookupService">
  <port name="personalInfoLookupPort"
        binding="tns:personalInfoLookupBinding">
    <iiop:address location="file:///objref.ior"/>
    <iiop:policy persistent="true"/>
    <iiop:policy serviceid="personalInfoLookup"/>
  </port>
</service>
```

Artix expects the IOR for the IIOP endpoint to be located in a file called `objref.ior`, and creates a persistent POA with an object id of `personalInfo` to configure the IIOP endpoint.

Using FTP

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

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

Adding an FTP Endpoint

You define an FTP endpoint using WSDL extensions that are placed within a the `port` element of a contract. The WSDL extensions provided by Artix allow you to specify a number of properties for establishing the FTP connection. In addition, they allow you to specify some of the properties used to define the naming properties for the files used by the transport.

Namespace

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

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

Defining the connection details

The connection details for the endpoint are defined in an `ftp:port` element.

The `ftp:port` element has two attributes: `host` and `port`.

The host attribute is required. It specifies the name of the machine hosting the FTP server to which the endpoint connects.

The port attribute is optional. It specifies the port number on which the FTP server is listening. The default value is 21.

Example 64 shows an example of a port element defining an FTP endpoint.

Example 64. Defining an FTP Endpoint

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

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

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

Specifying optional naming properties

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

The default naming implementation provided with Artix has two properties:

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

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

Coordinating Requests and Responses

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

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

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

Default implementation

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

Implementing the Client's Coordination Logic

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

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

The filename factory

The client-side filename factory is created by implementing the interface `com.iona.jbus.transports.ftp.policy.client.FilenameFactory`

Example 65 shows the interface.

Example 65. Client-Side Filename Factory Interface

```
package com.iona.jbus.transports.ftp.policy.client;

import javax.xml.namespace.QName;
import com.iona.webservices.wsdl.ext.ftp.FTPProperties;

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

  String getNextRequestFilename() throws Exception;
  String getRequestIncompleteFilename(String requestFilename)
  throws Exception;
  String getReplyFilename(String requestFilename)
  throws Exception;

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

The interface has four methods to implement:

`initialize()`

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

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

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

1. Determine if the user wants to use static filenames based on an `ftp:property` element in the contract. For more information see Using Properties to Control Coordination Behavior on page 195.
2. If so, it generates a static filename prefix for the requests.

3. If not, it uses the user supplied filename prefix for the requests.

`getNextRequestFilename()`

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

`getRequestIncompleteFilename()`

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

`getReplyFilename()`

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

The reply lifecycle policy

The reply lifecycle policy is created by implementing the com.iona.jbus.transports.ftp.policy.client.ReplyFileLifecycle interface. Example 66 shows the interface.

Example 66. Reply Lifecycle Interface

```
package com.iona.jbus.transports.ftp.policy.client;
public interface ReplyFileLifecycle
{
  boolean shouldDeleteReplyFile(String fileName)
  throws Exception;
  String renameReplyFile(String fileName)
  throws Exception;
}
```

The interface has two methods to implement:

`shouldDeleteReplyFile()`

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

`renameReplyFile()`

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

Configuring the client's coordination logic

If you choose to implement your own coordination logic for an FTP client endpoint, you need to configure the endpoint to load the your implementation classes. This is done by adding two configuration values to the endpoint's Artix configuration scope:

- `plugins:ftp:policy:client:filenameFactory` specifies the name of the class implementing the client's filename factory.
- `plugins:ftp:policy:client:replyFileLifecycle` specifies the name of the class implementing the client's reply lifecycle policy.

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

Example 67 shows an example of an Artix ESB configuration scope that specifies an FTP client endpoint's coordination policies.

Example 67. Configuring an FTP Client Endpoint

```
ftp_client
{
  plugins:ftp:policy:client:filenameFactory="demo.ftp.policy.client.
     myFilenameFactory";
  plugins:ftp:policy:client:replyFileLifecycle="demo.ftp.policy.client.
     myReplyFileLifecycle";
};
```

For more information on configuring Artix ESB see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

Implementing the Server's Coordination Logic

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

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

The filename factory

The server-side filename factory is created by implementing the interface `com.iona.jbus.transports.ftp.policy.server.FilenameFactory`. Example 68 shows the interface.

Example 68. Server-Side Filename Factory Interface

```
package com.iona.jbus.transports.ftp.policy.server; import

javax.xml.namespace.QName;

import com.iona.jbus.Bus;
import com.iona.transports.ftp.Element;
import com.iona.webservices.wsdl.ext.ftp.FTPProperties;
public interface FilenameFactory
{
  void initialize(Bus bus, QName service, String port,
                  FTPProperties properties) throws Exception;

  String getRequestFilenamesRegEx()
  throws Exception;
  Element[] updateRequestFiles(Element[] inElements)
  throws Exception;
  String getReplyIncompleteFilename(String requestFilename)
  throws Exception;
  String getReplyFilename(String requestFilename)
  throws Exception;
  FilenameFactoryPropertyMetaData[]
  getPropertiesMetaData();
}
```

The interface has six methods to implement:

`initialize()`

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

- the bus that has activated the transport.
- the QName of the service to which the endpoint is implementing.
- the value of the `name` attribute for the `port` element defining the endpoint’s connection details.

- an array containing any properties you specified as `ftp:property` elements in your server endpoint's contract.

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

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

`getRequestFileRegEx()`

`getRequestFileRegEx()` is called by the transport when it initializes the server-side FTP listener. It returns a regular expression that is used to match request filenames intended for a specific server instance. For example, the default implementation returns a regular expression of the form:

```
{wsdl:tns}_{wsdl:service(@name)}_{wsdl:port(@name
)}_{reqUuid}
```

`updateRequestFiles()`

`updateRequestFiles()` is called by the transport after it determines the list of possible requests and before it dispatches the requests to the service implementation for processing. It receives an array of `com.iona.transports.ftp.Element` objects. This array is a list of all the request messages selected by the request filename regular expression. `updateRequestFiles()` returns an array of `Element` objects containing only the messages that are to be dispatched to the service implementation.

`getReplyIncompleteFilename()`

`getReplyInclompleteFilename()` is called by the transport when it is ready to post a response. It receives the filename of the request that generated the response. It returns a string that is used as the filename for the response as it is being written to the FTP server. For example, the default implementation returns `_incomplete` appended to request filename.

getReplyFilename()

getReplyFilename() is called by the transport after it finishes writing a response to the FTP server. It receives the filename of the request that generated the response. It returns a string that is used as the filename for the completed response. For example, the default implementation returns _reply appended to request filename.

getPropertiesMetaData()

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

The request lifecycle policy

The request lifecycle policy is created by implementing the com.iona.jbus.transports.ftp.policy.server.RequestFileLifecycle interface. Example 69 shows the interface.

Example 69. Request Lifecycle Interface

```
package com.iona.jbus.transports.ftp.policy.server;
public interface RequestFileLifecycle
{
  boolean shouldDeleteRequestFile(String fileName)
  throws Exception;
  String renameRequestFile(String fileName)
  throws Exception;
}
```

The interface has two methods to implement:

shouldDeleteRequestFile()

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

renameRequestFile()

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

Configuring the server's coordination logic

If you choose to use your own coordination logic for an FTP server endpoint, you need to configure the endpoint to load the proper implementation classes. This is done by adding two configuration values to the endpoint's Artix configuration scope:

- `plugins:ftp:policy:server:filenameFactory` specifies the name of the class implementing the server's filename factory.
- `plugins:ftp:policy:server:requestFileLifecycle` specifies the name of the class implementing the server's request lifecycle policy.

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

Example 70 shows an example of an Artix configuration scope that specifies an FTP server endpoint's coordination policies.

Example 70. Configuring an FTP Server Endpoint

```
ftp_client
{
 plugins:ftp:policy:server:filenameFactory="demo.ftp.policy
  .server.myFilenameFactory";
  plugins:ftp:policy:server:requestFileLifecycle="demo.ftp.pol
  icy.client.myReqFileLifecycle";
};
```

For more information on configuring Artix see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

Using Properties to Control Coordination Behavior

In order to ensure that your FTP client endpoints and FTP server endpoints are using the same coordination behavior, you may need to pass some information to the transports as they initialize. To make this information available to both sides of the application and still be settable at run time, the Artix FTP transport allows you to provide custom properties that are settable in an endpoint's contract. These properties are set using the `ftp:properties` element.

Properties in the contract

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

For example, imagine an FTP endpoint defined by the port element in Example 71.

Example 71. FTP Endpoint with Custom Properties

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

The endpoint is configured using two custom FTP properties:

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

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

Supporting the properties

The initialize() method of both the client-side filename factory and the server-side filename factory take a com.iona.webservices.wsdl.ext.ftp.FTPProperties object. The FTPProperties object is populated by the contents of the endpoints ftp:properties element when the transport is initialized.

The FTPProperties object can be used to access all of the properties defined by ftp:property elements. To access the properties you do the following:

1. Use the getExtensors() method to get an Iterator object.
2. Using the Iterator objects next() method, get the elements in the list.
3. Cast the return value of the next() method to an FTPProperty object.

Each com.iona.webservices.wsdl.ext.ftp.FTPProperty object contains one name/value pair from one ftp:property element. You can extract the value of the name attribute using the FTPProperty object's getProperty() with the constant com.iona.webservices.wsdl.ext.ftp.FTPProperty.NAME. You can extract the value of the value attribute using the FTPProperty object's getProperty() with the constant com.iona.webservices.wsdl.ext.ftp.FTPProperty.VALUE.

Once you have the values of the properties, it is up to you to determine how they impact the coordination scheme.

Example 72 shows code for supporting the properties shown in Example 71.

Example 72. Using Custom FTP Properties

```
import com.iona.webservices.wsdl.ext.FTPProperties; import
com.iona.webservices.wsdl.ext.FTPProperty;
String nameTypeProp = "UseHumanNames"; String lastNameProp
= "LastName";
for (Iterator it = properties.getExtensors();
it.hasNext();)
{
  FTPProperty property = (FTPProperty)it.next(); String n =
  property.getProperty(FTPProperty.NAME);
  if (nameTypeProp.equals(n))
  {
    Boolean useHuman = new
    Boolean(property.getProperty(FTPProperty.VALUE));
  }
  if (lastNameProp.equals(n))
  {
    String lastName =
    property.getProperty(FTPProperty.VALUE);
  }
}
```

Filling in the filename factory property metadata

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

FilenameFactoryPropertyMetaData objects have three fields:

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

`FilenameFactoryPropertyMetaData` objects do not have any methods for populating its fields once the object is instantiated. You must set all of the values using the constructor that is shown in Example 73.

Example 73. Constructor for FilenameFactoryPropertyMetaData

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

Example 74 shows code for creating an array to be returned from getPropertiesMetaData().

Example 74. Populating the Filename Properties Metadata

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

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

Using WebSphere MQ

Artix can use WebSphere MQ to transport messages and leverage much of WebSphere's infrastructure to provide QoS.

Adding a WebSphere MQ Endpoint

The description for an Artix WebSphere MQ endpoint is entered in a `port` element of the Artix contract containing the interface to be exposed over WebSphere MQ. Artix defines two elements to describe WebSphere MQ endpoints and their attributes:

- `mq:client` defines an endpoint for a WebSphere MQ client application.
- `mq:server` defines an endpoint for a WebSphere MQ server application.

You can use one or both of the WebSphere MQ elements to describe a WebSphere MQ endpoint. Each can have different configurations depending on the attributes you choose to set.

WebSphere MQ namespace

The WSDL extensions used to describe WebSphere MQ transport details are defined in the WSDL namespace `http://schemas.iona.com/transports/mq`. If you are going to add a WebSphere MQ port by hand you will need to include the following in the `definitions` tag of your contract:

```
xmlns:mq="http://schemas.iona.com/transports/mq"
```

Required Attributes

When you define a WebSphere MQ endpoint you need to provide at least enough information for the endpoint to connect to its message queues. For any WebSphere application that means setting the `QueueManager` and `QueueName` attributes in the `port` element. In addition, if you are configuring a client that expects to receive replies from the server, you need to set the `ReplyQueueManager` and `ReplyQueueName` attributes of the `mq:client` element defining the client endpoint.

In addition, if you are deploying applications on a machine with a full MQ installation, you need to set the `Server_Client` attribute to `client` if the endpoint is going to use remote queues. This setting instructs Artix to load `libmqic` instead of `libmqm`.

Using the command line tool

To use **wsdltoservice** to add a WebSphere MQ endpoint use the tool with the following options.

```
wsdltoservice {-transport mq} [-e service] [-t port]
[-b binding] [-sqm queueManager] [-sqn queue] [-srqm
queueManager] [-srqn queue] [-smqn modelQueue] [-sus
usageStyle] [-scs correlationStyle] [-sam
accessMode] [-sto timeout] [-sme expiry] [-smp
priority] [-smi messageId] [-sci correlationId] [-sd
delivery] [-st transactional]

[-sro reportOption] [-sf format] [-sad
applicationData] [-sat accountingToken] [-scn
connectionName] [-sc convert] [-scr reusable] [-scfp
fastPath] [-said idData] [-saod originData] [-cqm
queueManager] [-cqn queue] [-crqm queueManager]
[-crqn queue] [-cmqn modelQueue]

[-cus usageStyle] [-ccs correlationStyle] [-cam
accessMode] [-cto timeout] [-cme expiry] [-cmp
priority] [-cmi messageId] [-cci correlationId] [-cd
delivery] [-ct transactional] [-cro reportOption]
[-cf format] [-cad applicationData] [-cat
accountingToken] [-ccn connectionName] [-cc convert]
[-ccr reusable] [-ccfp fastPath] [-caid idData]
[-caod originData] [-caqn queue] [-cui userId]
[-o file] [-d dir] [-L file] [[-quiet] | [-verbose]]
[-h] [-v] wsdlurl
```

The `-transport mq` flag specifies that the tool is to generate a WebSphere MQ service. The other options are as follows.

Table 29. Options for Adding a WebSphere MQ Endpoint

Option	Description
`-e` *service*	Specifies the name of the generated `service` element.
`-t` *port*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *binding*	Specifies the name of the binding for which the endpoint is generated.
`-sqm` *queueManager*	Specifies the name of the server's queue manager.
`-sqn` *queue*	Specifies the name of the server's request queue.
`-srqm` *queueManager*	Specifies the name of the server's reply queue manager.
`-srqn` *queue*	Specifies the name of the server's reply queue.
`-smqn` *modelQueue*	Specifies the name of the server's model queue.
`-sus` *usageStyle*	Specifies the value of the server's `UsageStyle` attribute. Valid values are `Peer`, `Requester`, or `Responder`.

Option	Description
`-scs` *`correlationStyle`*	Specifies the value of the server's `CorrelationStyle` attribute. Valid values are `messageId`, `correlationId`, or `messageId copy`.
`-sam` *`accessMode`*	Specifies the value of the server's `AccessMode` attribute. Valid values are `peek`, `send`, `receive`, `receive exclusive`, or `receive shared`.
`-sto` *`timeout`*	Specifies the value of the server's `Timeout` attribute.
`-sme` *`expiry`*	Specifies the value of the server's `MessageExpiry` attribute.
`-smp` *`priority`*	Specifies the value of the server's `MessagePriority` attribute.
`-smi` *`messageId`*	Specifies the value of the server's `MessageId` attribute.
-sci *`correlationId`*	Specifies the value of the server's `CorrelationID` attribute.
`-sd` *`delivery`*	Specifies the value of the server's `Delivery` attribute.
`-st` *`transactional`*	Specifies the value of the server's `Transactional` attribute. Valid values are `none`, `internal`, or `xa`.
`-sro` *`reportOption`*	Specifies the value of the server's `ReportOption` attribute. Valid values are `none`, `coa`, `cod`, `exception`, `expiration`, or `discard`.
`-sf` *`format`*	Specifies the value of the server's `Format` attribute.
`-sad` *`applicationData`*	Specifies the value of the server's `ApplicationData` attribute.
`-sat` *`accountingToken`*	Specifies the value of the server's `AccountingToken` attribute.
`-scn` *`connectionName`*	Specifies the name of the connection by which the adapter connects to the queue.
`-sc` *`convert`*	Specifies if the messages in the queue need to be converted to the system's native encoding. Valid values are `true` or `false`.
`-scr` *`reusable`*	Specifies the value of the server's `ConnectionReusable` attribute. Valid values are `true` or `false`.
`-scfp` *`fastPath`*	Specifies the value of the server's `ConnectionFastPath` attribute. Valid values are `true` or `false`.
`-said` *`idData`*	Specifies the value of the server's `ApplicationIdData` attribute.
`-saod` *`originData`*	Specifies the value of the server's `ApplicationOriginData` attribute.
`-cqm` *`queueManager`*	Specifies the name of the client's queue manager.
`-cqn` *`queue`*	Specifies the name of the client's request queue.
`-crqm` *`queueManager`*	Specifies the name of the client's reply queue manager.
`-crqn` *`queue`*	Specifies the name of the client's reply queue.

Option	Description
-cmqn *modelQueue*	Specifies the name of the client’s model queue.
-cus *usageStyle*	Specifies the value of the client’s UsageStyle attribute. Valid values are Peer, Requester, or Responder The default value is Requester.
-ccs *correlationStyle*	Specifies the value of the client’s CorrelationStyle attribute. Valid values are messageId, correlationId, or messageId copy.
-cam *accessMode*	Specifies the value of the client’s AccessMode attribute. Valid values are peek, send, receive, receive exclusive, or receive shared.
-cto *timeout*	Specifies the value of the client’s Timeout attribute.
-cme *expiry*	Specifies the value of the client’s MessageExpiry attribute.
-cmp *priority*	Specifies the value of the client’s MessagePriority attribute.
-cmi *messageId*	Specifies the value of the client’s MessageId attribute.
-cci *correlationId*	Specifies the value of the client’s CorrelationId attribute.
-cd *delivery*	Specifies the value of the client’s Delivery attribute.
-ct *transactional*	Specifies the value of the client’s Transactional attribute. Valid values are none, internal, or xa.
-cro *reportOption*	Specifies the value of the client’s ReportOption attribute. Valid values are none, coa, cod, exception, expiration, or discard.
-cf *format*	Specifies the value of the client’s Format attribute.
-cad *applicationData*	Specifies the value of the client’s ApplicationData attribute.
-cat *accountingToken*	Specifies the value of the client’s AccountingToken attribute.
-ccn *connectionName*	Specifies the name of the connection by which the adapter connects to the queue.
-cc *convert*	Specifies if the messages in the queue need to be converted to the system’s native encoding. Valid values are true or false.
-ccr *reusable*	Specifies the value of the client’s ConnectionReusable attribute. Valid values are true or false.
-ccfp *fastPath*	Specifies the value of the client’s ConnectionFastPath attribute. Valid values are true or false.
-caid *idData*	Specifies the value of the client’s ApplicationIdData attribute.
-caod *originData*	Specifies the value of the client’s ApplicationOriginData attribute.

Option	Description
-caqn *queue*	Specifies the remote queue to which a server will put replies if its queue manager is not on the same host as the client's local queue manager.
-cui *userId*	Specifies the value of the client's UserIdentification attribute.
-o *file*	Specifies the filename for the generated contract. The default is to append –service to the name of the imported contract.
-d *dir*	Specifies the output directory for the generated contract.
-L *file*	Specifies the location of your Artix license file. The default behavior is to check IT_PRODUCT_DIR\etc\license.txt.
-quiet	Specifies that the tool runs in quiet mode.
-verbose	Specifies that the tool runs in verbose mode.
-h	Displays the tool's usage statement.
-v	Displays the tool's version.

For more information about the specific attributes and their values see the ***Artix WSDL Extension Reference***.

Example

An Artix contract exposing an interface, monsterBash, bound to a SOAP payload format, Raydon, on an WebSphere MQ queue, UltraMan would contain a service element similar to Example 75.

Example 75. Sample WebSphere MQ Port

```
<service name="Mothra">
  <port name="X" binding="tns:Raydon">
    <mq:server QueueManager="UMA"
               QueueName="UltraMan" ReplyQueueManager="WINR"
               ReplyQueueName="Elek" AccessMode="receive"
               CorrelationStyle="messageId copy"/>
</port>
</service>
```

WebSphere MQ Connection Settings

The Artix ESB MQ transport makes some basic connection decisions. These have an impact on the privileges required when Artix ESB connects to a queue manager, and how context information is handled.

Granting authority for setting context information

You can control access privileges using the WebSphere MQ **setmqaut** command.

Example 76 shows the commands required to provide the proper authorization for the Artix ESB transport. For example, these commands apply when the Artix AccessMode WSDL extension is set to send+setid.

Example 76. Granting the MQ Transport Authorization

```
setmqaut -m MY_QMNGR -t queue -n MY_Q -g grp +all
```

```
setmqaut -m MY_QMNGR -t queue -n MY_Q -g grp -setall
-passid -passall
```

For more details on setting authorizations, see the entry for MQ AccessModes in the ***Artix WSDL Extension Reference***.

Further information

For more information on the implications of using MQOO_SET_ALL_CONTEXT and the **setmqaut** command, see:

http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp?topic=/com.ibm.mq.amqzag.doc/fa15980_.htm

Specifying the WebSphere Library to Load

The version of the WebSphere MQ shared library loaded by an Artix MQ endpoint alters the types of queues that an endpoint can access. For example, if an Artix endpoint loads the MQ client shared library, it will only be able to use queues hosted on a remote machine. Artix provides an attribute in the MQ WSDL extensions that allows you to control which library is loaded.

The attribute

Both the mq:server element and the mq:client element support the attribute that is used to specify which MQ libraries to load. The Server_Client attribute specifies which shared libraries to load on systems with a full WebSphere MQ installation. Table 30 describes the settings for this attribute for each type of WebSphere MQ installation.

Table 30. WebSphere MQ Server_Client Attribute Settings

MQ Installation	Server_Client Value	Behavior
Full		The server shared library (libmqm) is loaded and the application will use queues hosted on the local machine.
Full	server	The server shared library (libmqm) is loaded and the application will use queues hosted on the local machine.
Full	client	The client shared library (libmqic) is loaded and the application will use queues hosted on a remote machine.

MQ Installation	Server_Client Value	Behavior
Client		The application will attempt to load the server shared library (libmqm) before loading the client shared library (libmqic). The application accesses queues hosted on a remote machine.
Client	server	The application will fail because it cannot load the server shared libraries.
Client	client	The client shared library (libmqic) is loaded and the application accesses queues hosted on a remote machine.

Example

Example 77 shows a service element for an MQ endpoint that uses the MQ client shared library.

Example 77. ARTIX MQ Endpoint Using MQ Client Library

```
<service name="Mothra">
  <port name="X" binding="tns:Raydon">
    <mq:server QueueManager="UMA"
               QueueName="UltraMan" ReplyQueueManager="WINR"
               ReplyQueueName="Elek" Server_Client="client" />
  </port>
</service>
```

Using Queues on Remote Hosts

When interoperating between WebSphere MQ endpoints whose queue managers are on different hosts, Artix requires that you specify the name of the remote queue to which the server will post reply messages. This ensures that the server will put the replies on the proper queue. Otherwise, the server will receive a request message with the ReplyToQ field set to a queue that is managed by a queue manager on a remote host and will be unable to send the reply.

You specify this server's local reply queue name in the mq:client element's AliasQueueName attribute when you define it in the client endpoint's contract.

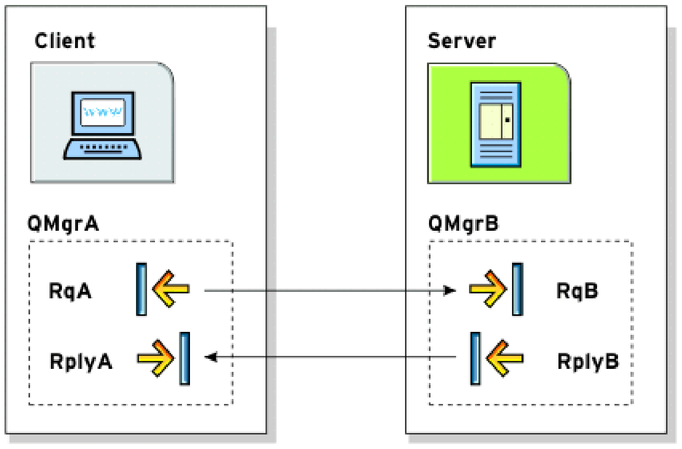
Effect of AliasQueueName

When you specify a value for the AliasQueueName attribute in an mq:client element, you alter how Artix populates the request message's ReplyToQ field and ReplyToQMgr field. Typically, Artix populates the reply queue information in the request message's message descriptor with the values specified in the ReplyQueueManager attribute and the ReplyQueueName attribute. Setting the AliasQueueName attribute causes Artix to leave ReplytoQMgr empty, and to set ReplyToQ to the value of the AliasQueueName attribute. When the ReplyToQMgr field of the message descriptor is left empty, the sending queue manager inspects the queue named in the ReplyToQ field to determine who its queue manager is and uses that value for ReplyToQMgr. The server puts the message on the remote queue that is configured as a proxy for the client's local reply queue.

Example

If you had a system defined similar to that shown in Figure 2, you would need to use the AliasQueueName attribute setting when configuring your WebSphere MQ client. In this set up the client is running on a host with a local queue manager QMgrA. QMgrA has two queues configured. RqA is a remote queue that is a proxy for RqB and RplyA is a local queue. The server is running on a different machine whose local queue manager is QMgrB. QMgrB also has two queues. RqB is a local queue and RplyB is a remote queue that is a proxy for RplyA. The client places its request on RqA and expects replies to arrive on RplyA.

Figure 2. MQ Remote Queues



The port elements for the client and server for this deployment are shown in Example 78. The AliasQueueName attribute is set to RplyB because that is the remote queue proxying for the reply queue in server's local queue manager. The ReplyQueueManager attribute and the ReplyQueueName attribute are set to the client's local queue manager so that it knows where to listen for responses. In this example, the server's ReplyQueueManager attribute and ReplyQueueName attribute do not need to be set because you are assured that the client is populating the request's message descriptor with the needed information for the server to determine where replies are sent.

Example 78. Setting Up WebSphere MQ Ports for Intercommunication

```
<mq:client QueueManager="QMgrA" QueueName="RqA"
        ReplyQueueManager="QMgrA" ReplyQueueName="RplyA"
        AliasQueueName="RplyB"
        Format="string" Convert="true"/>
<mq:server QueueManager="QMgrB" QueueName="RqB"
        Format="String" Convert="true"/>
```

Setting a Value of the Message Descriptor's Format Field

WebSphere MQ messages have a `Format` field in their message descriptors. Message receivers use this field to determine the nature of the data in the nature. What the message receiver does with this information is the responsibility of the application developer. Artix, however, uses the `Format` field to determine if the contents of a message are to undergo codeset conversion.

You can specify the value placed in the message descriptor's `Format` field using the `Format` attribute. This attribute is supported by both the `mq:client` element and the `mq:server` element and its value is a string specifying the name of the message's format.

Special values

The `Format` attribute can take the special values described in Table 31.

Table 31. WebSphere MQ Format Attribute Settings

Attribute Setting	Description
`none` (Default)	Corresponds to MQFMT_NONE. No format name is specified.
`string`	Corresponds to MQFMT_STRING. `string` specifies that the message consists entirely of character data. The message data may be either single-byte characters or double-byte characters.
`unicode`	Corresponds to MQFMT_STRING. `unicode` specifies that the message consists entirely of Unicode characters. (Unicode is not supported in Artix at this time.)
`event`	Corresponds to MQFMT_EVENT. `event` specifies that the message reports the occurrence of a WebSphere MQ event. Event messages have the same structure as programmable commands.
`programmable command`	Corresponds to MQFMT_PCF. `programmable` command specifies that the messages are user-defined messages that conform to the structure of a programmable command format (PCF) message.

For more information, consult the IBM Programmable Command Formats and Administration Interfaces documentation at http://publibfp.boulder.ibm.com/epubs/html/csqzac03/csqzac030d.htm#Header_12.

Using codeset conversion

Artix uses the value of the `Format` field in an MQ message header to determine

if the message data should be converted into a host systems native codeset. If the `Format` field is set to `MQFMT_STRING`, Artix will attempt to convert the data into the host's native codeset. If the `Format` field has any other value, Artix will not attempt to perform codeset conversion.

If you are interoperating with systems that use a different codeset than the system your endpoint is hosted on, you need to set the `Format` attribute of the Artix endpoint to string. This is particularly important when you are interoperating with WebSphere MQ applications hosted on a mainframe because the data needs to be converted into the systems native data format. Not doing so will result in the mainframe receiving corrupted data.

Example

Example 79 shows an `mq:client` element that defines an endpoint used for making requests against a server on a mainframe system. In this particular example, we are talking directly to the mainframe queue manager.

Example 79. WebSphere MQ Client Talking to the Mainframe

```
<mq:client QueueManager="Mainframe_Request_Queue_Manager"
         QueueName="Application_request_queue_name"
         ReplyQueueManager="Mainframe_Reply_Queue_Manager"
         ReplyQueueName="Application_reply_queue_name"
         Server_Client="client" Format="string" Convert=
"true" />
```

In this example, you will also need to set the `MQSERVER` environment variable. The section *Using a Remote MQ Server* in ***Artix Technical Use Cases*** explains this in detail.

Using Tuxedo

Artix allows services to connect using Tuxedo's transport mechanism. This provides them with all of the qualities of service associated with Tuxedo.

Tuxedo namespaces

To use the Tuxedo transport, you need to describe the endpoint using Tuxedo in the physical part of an Artix contract. The extensions used to describe a Tuxedo endpoint are defined in the following namespace:

```
xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
```

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

Defining the Tuxedo services

As with other transports, the Tuxedo transport description is contained within a `port` element. Artix uses a `tuxedo:server` element to describe the attributes of a Tuxedo endpoint. The `tuxedo:server` element has a child element, `tuxedo:service`, that gives the bulletin board name of a Tuxedo endpoint. The bulletin board name for the endpoint is specified in the element's `name` attribute. You can define more than one Tuxedo service to act as an endpoint.

Mapping operations to a Tuxedo service

For each of the Tuxedo services that are endpoints, you must specify which of the operations bound to the endpoint being defined are handled by the Tuxedo service. This is done using one or more `tuxedo:input` child elements. The `tuxedo:input` element takes one required attribute, `operation`, that specifies the WSDL operation that is handled by this Tuxedo service endpoint.

Using the command line tools

To use **wsdltoservice** to add a Tuxedo endpoint use the tool with the following options.

```
wsdltoservice {-transport tuxedo} [-e service] [-t
port] [-b binding] [-tsn tuxService] [-tfn
tuxService:tuxFunction] [-ton

tuxService:operation] [-o file] [-d dir] [-L file]
[[-quiet] | [-verbose]] [-h] [-v] wsdlurl
```

The `-transport tuxedo` flag specifies that the tool is to generate a Tuxedo service. The other options are as follows.

Table 34. Options for Adding a Tuxedo Service

Option	Description
-e *service*	Specifies the name of the generated service element.
-t *port*	Specifies the value of the name attribute of the generated port element.
-b *binding*	Specifies the name of the binding for which the endpoint is generated.
-tsn *tuxService*	Specifies the name the service uses when registering with the Tuxedo bulletin board.
-tfn *tuxService*:*tuxFunction*	Specifies the name of the function to be used on the specified Tuxedo bulletin board.
-ton *tuxService*:*operation*	Specifies the WSDL operation that is handled by the specified Tuxedo endpoint.
-o *file*	Specifies the filename for the generated contract. The default is to append -service to the name of the imported contract.
-d *dir*	Specifies the output directory for the generated contract.
-L *file*	Specifies the location of your Artix license file. The default behavior is to check IT_PRODUCT_DIR\etc\license.txt.
-quiet	Specifies that the tool runs in quiet mode.
-verbose	Specifies that the tool runs in verbose mode.
-h	Displays the tool's usage statement.
-v	Displays the tool's version.

For more information about the specific attributes and their values see the ***Artix WSDL Extension Reference***.

Example

An Artix contract exposing the personalInfoService as a Tuxedo endpoint would contain a service element similar to Example 80.

Example 80. Tuxedo Port Description

```
<service name="personalInfoService">
  <port binding="tns:personalInfoBinding"
  name="tuxInfoPort">
    <tuxedo:server>
      <tuxedo:service name="personalInfoService">
<tuxedo:input operation="infoRequest"/>
      </tuxedo:service>
    </tuxedo:server>
  </port>
</service>
```

Part III

Other Artix ESB Features

In this part

This part contains the following chapters:

Working with CORBA

CORBA, unlike the other platforms supported by Artix ESB, specifies both a mapping between the logical messages and a network protocol. Because these two cannot be decoupled, Artix provides extensions for both and requires that they be used together. To further enforce the coupling of the CORBA payload format and the CORBA network protocol all Artix tools that generate CORBA extensions generate them in sets.

Adding a CORBA Binding

CORBA applications use a specific payload format when making and responding to requests. The CORBA binding, described using a WSDL extension, specifies the repository ID of the IDL interface represented by the port type, resolves parameter order and mode ambiguity in the operations' messages, and maps the XML Schema data types to CORBA data types.

In addition to the binding information, Artix also uses a `corba:typemap` element to unambiguously describe how data is mapped to CORBA data types. For primitive types, the mapping is straightforward. However, complex types such as structures, arrays, and exceptions require more detailed descriptions. For a detailed description of the CORBA type mappings see ***Artix for CORBA***.

Options

To add a CORBA binding to an Artix contract you can choose one of the following methods:

- Use the **wsdltocorba** command line tool. The command line tool automatically generates the binding and type map information for a specified port type. See Using wsdltocorba.
- Enter the binding and typemap information by hand using a text editor or XML editor. This option provides you the flexibility to customize the binding. However, hand editing Artix contracts can be a time consuming process and provides no error checking mechanisms. For information on the WSDL extensions used to specify a CORBA binding see Mapping to the binding.

Using wsdltocorba

The **wsdltocorba** tool adds CORBA binding information to an existing Artix contract. To generate a CORBA binding use the following command:

```
wsdltocorba {-corba} {-i portType} [-d dir] [-b
binding] [-o file] [-props namespace] [-wrapped] [-L
file] [[-quiet] | [-verbose]] [-h] [-v] wsdl_file
```

The command has the following options:

Table 35. Options for Adding a CORBA Binding

Option	Description
`-corba`	Instructs the tool to generate a CORBA binding for the specified port type.
`-i` *`portType`*	Specifies the name of the port type being mapped to a CORBA binding.
`-d` *`dir`*	Specifies the directory into which the new WSDL file is written.
`-b` *`binding`*	Specifies the name for the generated CORBA binding. Defaults to *`portType`*`Binding`.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl_file`*`-corba.wsdl`.
`-props` *`namespace`*	Specifies the namespace to use for the generated CORBA typemap
`-wrapped`	Specifies that the generated CORBA binding uses wrapper types.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-quiet`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-verbose`	Specifies that the tool runs in verbose mode.
`-h`	Specifies that the tool will display a usage message.
`-v`	Displays the tool's version.

The generated WSDL file will also contain a CORBA endpoint with no address specified.

WSDL namespace

The WSDL extensions used to describe CORBA data mappings and CORBA transport details are defined in the WSDL namespace `http://schemas.iona.com/bindings/corba`. To use the CORBA extensions you will need to include the following in the `definitions` tag of your contract:

```
xmlns:corba="http://schemas.iona.com/bindings/corba"
```

Mapping to the binding

The extensions used to map a logical operation to a CORBA binding are described in detail below:

- `corba:binding`

  `corba:binding` indicates that the binding is a CORBA binding. This element has one required attribute: `repositoryID`. The `repositoryID` attribute specifies the full type ID of the interface. The type ID is embedded in the object's IOR and therefore must conform to the IDs that are generated from an IDL compiler. These are of the form:

  ```
  IDL:module/interface:1.0
  ```

  The `corba:binding` element also has an optional attribute, bases, that specifies that the interface being bound inherits from another interface. The value for bases is the type ID of the interface from which the bound interface inherits. For example, the following IDL:

  ```
  //IDL
  interface clash{}; interface bad : clash{};
  ```

  would produce the following `corba:binding`:

  ```
  <corba:binding repositoryID="IDL:bad:1.0"
              bases="IDL:clash:1.0"/>
  ```

- `corba:operation`

  `corba:operation` is an Artix-specific element of the `operation` element and describes the parts of the operation's messages. `corba:operation` takes a single attribute, name, which duplicates the name given in `operation`.

- `corba:param`

  `corba:param` is a child of `corba:operation`. Each `part` element of the input and output messages specified in the logical operation, except for the part representing the return value of the operation, must have a corresponding `corba:param` element. The parameter order defined in the binding must match the order specified in the IDL definition of the operation. The `corba:param` element has the following required attributes:

Table 36. Attributes of `corba:param`

Attribute	Description
`mode`	Specifies the direction of the parameter. The values directly correspond to the IDL directions: `in`, `inout`, `out`. Parameters set to `in` must be included in the input message of the logical operation. Parameters set to `out` must be included in the output message of the logical operation. Parameters set to `inout` must appear in both the input and output messages of the logical operation.
`idltype`	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types, and corbatm: for complex data types, which are mapped out in the `corba:typeMapping` portion of the contract.
`name`	Specifies the name of the parameter as given in the logical portion of the contract.

- `corba:return`

  `corba:return` is a child of `corba:operation` and specifies the return type, if any, of the operation. It has two attributes:

Attribute	Description
`name`	Specifies the name of the parameter as given in the logical portion of the contract.
`idltype`	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types and corbatm: for complex data types which are mapped out in the `corba:typeMapping` portion of the contract.

- `corba:raises`

  `corba:raises` is a child of `corba:operation` and describes any exceptions the operation can raise. The exceptions are defined as fault messages in the logical definition of the operation. Each fault message must have a corresponding `corba:raises` element. `corba:raises` has one required attribute, `exception`, which specifies the type of data returned in the exception.

In addition to operations specified in `corba:operation` tags, within the WSDL `operation` element, each `operation` element in the binding must also specify empty `input` and `output` elements as required by the WSDL specification. The CORBA binding specification, however, does not use them.

For each fault message defined in the logical description of the operation, a corresponding `fault` element must be provided in the operation element, as required by the WSDL specification. The name attribute of the fault element specifies the name of the schema type representing the data passed in the fault message.

Example

For example, a logical interface for a system to retrieve employee information might look similar to personalInfoLookup, shown in Example 81.

Example 81. personalInfo lookup port type

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
</message>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo"/>
</message>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound"/>
</message>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest"/>
    <output name="return"
            message="personalLookupResponse"/>
    <fault name="exception" message="idNotFoundException"/>
  </operation>
</portType>
```

The CORBA binding for personalInfoLookup is shown in Example 82.

Example 82. personalInfoLookup CORBA Binding

```
<binding name="personalInfoLookupBinding"
       type="tns:personalInfoLookup">
  <corba:binding
  repositoryID="IDL:personalInfoLookup:1.0"/>
  <operation name="lookup">
    <corba:operation name="lookup">
      <corba:param name="empId" mode="in"
idltype="corba:long"/>
      <corba:return name="return"
                idltype="corbatm:personalInfo"/>
      <corba:raises exception="corbatm:idNotFound"/>
    </corba:operation>
    <input/>
    <output/>
    <fault name="personalInfoLookup.idNotFound"/>
</operation>
</binding>
```

Creating a CORBA Endpoint

Generally, when you are creating a CORBA endpoint with Artix, you need to do two things:

- Specify the port information in the Artix contract so that Artix can instantiate the appropriate port.
- Generate the IDL describing your service so that a native CORBA application can understand the interfaces of the new service.

Configuring an Artix CORBA Endpoint

CORBA endpoints are described using the Artix-specific WSDL elements `corba:address` and `corba:policy` within the WSDL `port` element, to specify how a CORBA object is exposed.

Namespace

The namespace under which the CORBA extensions are defined is `http://schemas.iona.com/bindings/corba`. If you are going to add a CORBA endpoint by hand you will need to add this to your contract's `definition` element.

CORBA address specification

The IOR of the CORBA object is specified using the `corba:address` element.

You have four options for specifying IORs in Artix contracts:

- Specify the object's IOR directly in the contract, using the stringified IOR format:

  ```
  IOR:22342...
  ```

- Specify a file location for the IOR, using the following syntax:

  ```
  file:///file_name
  ```

NOTE: The file specification requires three backslashes (///).

- Specify that the IOR is published to a CORBA name service, by entering the object's name using the `corbaname` format:

  ```
  corbaname:rir/NameService#object_name
  ```

For more information on using the name service with Artix see ***Artix for CORBA***.

- Specify the IOR using `corbaloc`, by specifying the port at which the endpoint exposes itself, using the `corbaloc` syntax.

  ```
  corbaloc:iiop:host:port/service_name
  ```

  When using `corbaloc`, you must be sure to configure your endpoint to start up on the specified host and port.

Specifying POA policies

Using the optional `corba:policy` element, you can describe a number of POA polices the endpoint will use when creating the POA for connecting to a CORBA application. These policies include:

- the name of the generated POA
- if persistence is used
- the ID of the generated POA

Setting these policies lets you exploit some of the enterprise features of Micro Focus' Orbix 6.x, such as load balancing and fault tolerance, when deploying an Artix integration project. For information on using these advanced CORBA features, see the Orbix documentation.

POA name

Artix POAs are created with the default name of `WS_ORB`. To specify the name of the POA Artix creates to connect with a CORBA object, you use the following:

```
<corba:policy poaname="poa_name"/>
```

Persistence

By default Artix POAs have a persistence policy of `false`. To set the POA's persistence policy to `true`, use the following:

```
<corba:policy persistent="true"/>
```

ID assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by the ORB. To specify that the POA connecting a specific object should use a user-assigned ID, use the following:

```
<corba:policy serviceid="POAid"/>
```

This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Using the command line tool

You can use the **wsdltoservice** command line tool to add a CORBA endpoint definition to an Artix contract. To use **wsdltoservice** to add a CORBA endpoint use the tool with the following options.

```
wsdltoservice {-transport corba} [-e service] [-t
port] [-b binding] [-a address] [-poa poaName] [-sid
serviceId] [-pst persists] [-o file] [-d dir] [-L
file] [[-q] | [-V]] [-h] wsdlurl
```

The `-transport corba` flag specifies that the tool is to generate a CORBA endpoint. The other options are as follows.

Table 37. Options for Adding a CORBA Endpoint

Argument	Descriptions
`-e` *`service`*	Specifies the name of the generated `service` element.
`-t` *`port`*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *`binding`*	Specifies the name of the binding for which the service is generated.
`-a` *`address`*	Specifies the value used in the `corba:address` element of the port.
`-poa` *`poaName`*	Specifies the value of the POA name policy.
`-sid` *`serviceId`*	Specifies the value of the ID assignment policy.
`-pst` *`persists`*	Specifies the value of the persistence policy. Valid values are `true` and `false`.
`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *`dir`*	Specifies the output directory for the generated contract.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Example

For example, a CORBA port for the personalInfoLookup binding would look similar to Example 83:

Example 83. CORBA personalInfoLookup Port

```
<service name="personalInfoLookupService">
  <port name="personalInfoLookupPort"
        binding="tns:personalInfoLookupBinding">
    <corba:address location="file:///objref.ior"/>
    <corba:policy persistent="true"/>
    <corba:policy serviceid="personalInfoLookup"/>
  </port>
</service>
```

Artix expects the IOR for the CORBA object to be located in a file called objref.ior, and creates a persistent POA with an object id of personalInfo to connect the CORBA application.

Generating CORBA IDL

Artix clients that use a CORBA transport require that the IDL defining the interface exists and be accessible. Artix provides tools to generate the required IDL from an existing WSDL contract. The generated IDL captures the information in the logical portion of the contract and uses that to generate the IDL interface. Each portType element in the contract generates an IDL module.

From the command line

The **wsdltocorba** tool compiles Artix contracts and generates IDL for the specified CORBA endpoint. To generate IDL use the following command:

```
wsdltocorba { -idl } {-b binding} [-corba] [-i
portType] [-d dir] [-o file] [-L file] [[-q] | [-V]]
[-h] wsdl_file
```

The command has the following options:

Table 38. Options for Generating IDL

Option	Description
-idl	Instructs the tool to generate an IDL file from the specified binding.
-b *binding*	Specifies the CORBA binding from which IDL is to be generated.
-corba	Instructs the tool to generate a CORBA binding for the specified port type.
-i *portType*	Specifies the name of the port type being mapped to a CORBA binding.

Option	Description
`-d` *`dir`*	Specifies the directory into which the new WSDL file is written.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl file`*`.idl`.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-V`	Specifies that the tool runs in verbose mode.
`-h`	Specifies that the tool will display a usage message.

By combining the `-idl` and `-corba` flags with **wsdltocorba**, you can generate a CORBA binding for a logical operation and then generate the IDL for the generated CORBA binding. When doing so, you must also use the `-i` *`portType`* flag to specify the port type from which to generate the binding and the `-b` *`binding`* flag to specify the name of the binding from which to generate the IDL.

Using the Artix Transformer

The Artix transformer allows you to perform message transformations, data validation, and interface versioning without having to write additional code.

Using the Artix Transformer as a Service

Using the Artix transformer, you can create a Web service that does simple tasks such as converting dates into the proper format or generating HTML output without writing any code. You can also develop services to validate the format of requests before they are sent to a busy server for processing.

The data processing is performed by the Artix transformer which uses an XSLT script to determine how to process the data.

Procedure

To use the Artix transformer as a service you:

1. Define the data, interface, binding, and transport details for the server in an Artix contract.
2. Write the XSLT script that defines the data processing you want the transformer to perform.
3. Configure the service with the transformer's configuration details.

Defining the server

The contract for a service that is implemented by the Artix transformer is the same as the Artix contract for any other service in Artix. You need to define the complex types, if any, that the service uses. Then you need to define the messages used by the service to receive and respond to requests.

Once the data types and messages are defined, you then define the service's interface. The only limitation for a service that is implemented by the Artix Transformer is that it cannot have any fault messages. The interface can define multiple operations. Each operation will be processed using different XSLT scripts.

After defining the logical details of the service, you need to define the binding and network details for the service. The transformer can use any of the bindings and transports supported by Artix.

Writing the scripts

The XSLT scripts tell the transformer what it needs to do to process the data it receives. The scripts can be as simple or complex as they need to be to perform the task. The only requirement is that they are valid XSLT documents.

For more information about writing XSLT scripts read Writing XSLT Scripts.

Configure the transformer

The Artix transformer is an Artix plug-in and can be loaded by an Artix process.

This provides a great deal of flexibility in how you configure and deploy the process. There are two common deployment patterns for deploying the Artix transformer as a service. The first is to configure the transformer to load into the Artix container. The second is to configure the transformer to load directly into the client process which is making requests against it.

For a detailed discussion of how to configure and deploy the Artix Transformer see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

Using Artix to Facilitate Interface Versioning

One of the most common and difficult problems faced in large scale client server deployments is upgrading systems. For example, if you change the interface for your server to add new functionality or streamline communications, you then need to change all of the clients that access the server. This can mean upgrading thousands of clients that may be scattered across the globe.

The Artix transformer provides a solution to this problem that allows you to slowly upgrade the clients without disrupting their ability to function. Using the transformer you can develop an XSLT script that converts messages between the different interfaces. Then you can place the transformer between the old clients and the new server. This solution eliminates the need for operating two versions of the same server, or trying to do a massive client and server upgrade. It also does this without requiring you to do any custom programing.

Procedure

To use the Artix transformer for interface versioning do the following:

1. Create a composite Artix contract defining both versions of the interfaces that need to be supported.

2. Define an interface for the transformer that defines operations for mapping the interfaces.
3. Add a SOAP binding to the contract for the transformer's interface.
4. Add an HTTP port to the contract to define how the transformer can be contacted.
5. Write the XSLT scripts that define the message transformations.
6. Configure the transformer.
7. Configure the Artix chain builder to create a chain containing the transformer and the server on which clients will make requests.

Creating a composite contract

While the server and the client applications can be run without knowledge of the other's interface, the transformer responsible for translating the messages between to the two interface versions must know about all of the interface versions used. This includes all data type definitions and message definitions used by both versions of the interface.

You can create this composite contract in several ways. The most straightforward way is to create a new contract which imports both the new interface's contract and the old interface's contract. To import the contracts you place an `import` element for each contract just after the `definitions` element in the new contract and before any other elements in the new contract. The `import` element has two attributes. `location` specifies the pathname of the file containing the contract that is being imported. `namespace` defines the XML namespace under which the imported contract can be referenced.

For example, if you were creating a composite contract for interface versioning you would have two contracts; one for the server with the updated interface and one for the client using the legacy interface. The file name for the server's contract is `r2e2.wsdl` and the contract for the client is `r2e1.wsdl`. For simplicity, they are located in the same directory as the composite contract. The composite contract importing both versions of the interface is shown in Example 84.

Example 84. Composite WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="transformer"
       targetNamespace="http://www.widgets.com/transformer"
       xmlns="http://schemas.xmlsoap.org/wsdl/"
       xmlns:r1="http://www.widgets.com/r2e2Server"
       xmlns:r2="http://www.widgets.com/r2e1Client"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:tns="http://www.widgets.com/transformer"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<import location="r2e2.wsdl"
       namespace="http://www.widgets.com/r2e2Server/>
<import location="r2e1.wsdl"
       namespace="http://www.widgets.com/r2e1Client"/>
</definitions>
```

Note that in the `definitions` element of the contract, XML namespace shortcuts are defined for the imported contracts namespace. This makes using items defined in the imported contracts much easier.

Define the transformer's interface

Once you have imported all versions of the interface that you need to support into the transformer’s composite contract, you need to define the transformer’s interface. The transformer must have one operation defined for each transformation that is required to support all of the interface versions. For example, if you only changed the structure of the request message in when upgrading the server’s interface, the transformer only needs one operation because the transformation is only one way. If you changed both the request and response messages, the transformer’s interface will need two operations; one for the request message and one for the response.

The operation to transform a request from the client to the proper format for the server takes the client’s message as its `input` element and the server’s message as its `output` message. The operation to transform a response from the server to the proper format for a client takes the server’s outgoing message as its `input` element and the client’s incoming message as its `output` element.

NOTE: Fault messages are not supported.

When adding the operations, be sure to use the proper namespaces when referencing the messages for the different versions of the interface. Using the wrong namespaces could result in an invalid contract at the very least. If the contract is valid, and the namespaces are incorrect, your system will behave erratically.

For example, if the interface in Example 84 was updated so that both the client's request and the server's response need to be transformed the transformer's interface would need two operations. In this example the name of the request message is `widgetRequest` and the name of the response message is `widgetResponse`. The interface for the transformer, `versionTransform`, is shown in Example 85.

Example 85. Versioning Interface

```
<portType name="versionTransform">
  <operation name="requestTransform">
    <input name="oldRequest" message="r1:widgetRequest"/>
    <output name="newRequest" message="r2:widgetRequest"/>
  </operation>
  <operation name="responseTransform">
    <input name="newResponse" message="r2:widgetResponse"/>
    <output name="oldReponse" message="r1:widgetResponse"/>
  </operation>
</portType>
```

In the operation transforming the request, `requestTransform`, the input message is taken from the namespace **r1** which is the namespace under which the client's contract is imported. The output message is taken from **r2** which is the namespace under which the server's contract is imported. For the response message transformation, `responseTransform`, the order is reversed. The input message is from **r2** and the output message is from **r1**.

Defining the physical details for the transformer

After defining the operations used in transforming between the different version of the interface, you need to define the binding and network details for the transformer. The transformer can use any of the bindings and transports supported by Artix. For information on adding a binding for the transformer read *Understanding Bindings in WSDL*. For information on adding network details for the transformer read *Understanding How Endpoints are Defined in WSDL*.

Writing the XSLT scripts

The XSLT scripts tell the transformer what it needs to do to process the data it receives. The scripts can be as simple or complex as they need to be to perform the task. The only requirement is that they are valid XSLT documents. For more information about writing XSLT scripts read Writing XSLT Scripts.

Configuring the transformer

The Artix transformer is an Artix plug-in and can be loaded by an Artix process.

This provides a great deal of flexibility in how you configure and deploy the process. For a detailed discussion of how to configure and deploy the Artix transformer see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

When using the transformer to do interface versioning, you need to deploy it as part of a service chain. To build a service chain in Artix you deploy the Artix chain builder. Like the transformer, the chain builder is an Artix plug-in and provides a number of deployment options. One way of deploying the chain builder along with the transformer is to deploy it alongside the transformer in an Artix container.

For a detailed discussion of how to configure and deploy the Artix chain builder see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

WSDL Messages and the Transformer

Conceptually, the Artix transformer works on XML representations of the data passed along the wire. Your XSLT scripts are written based on the WSDL descriptions of the message's being processed. This relieves you of the burden of understanding how the data on the wire is represented.

Incoming messages

The virtual XML document the transformer uses as input is created by using the Artix contract to map the raw data from the input port into a DOM facade. The mapping is done as follows:

- If the message is defined using the doc-literal styles, the transformer uses the message part's schema definition to create a representation of the message.
- If the message is not defined using the doc-literal style, the transformer does the following to build an XML representation of the message:
    1. the name of the message's root element is the QName of the `message` element referred to by the operation's `input` element.
    2. Each `part` element in the input message is placed in an element derived from the `name` attribute of the `part` element.
    3. If the part is of a complex type, or an element of a complex type, the type's elements appear inside of the element containing the part.

For example, if you had a service defined by the WSDL fragment in Example 86 and the transformer implemented the operation configure, the XML document would be constructed using the message oldClientInput, which is the input message.

Example 86. WSDL Fragment for Transformer

```
<definitions targetNamespace="vehicle.demo.example"
          xmlns:tns="vehicle.demo.example"
          ...>
<types ...>
...
  <complexType name="vehicleType">
    <element name="vin" type="xsd:string" />
    <element name="model" type="xsd:string" />
  </complexType>
</types>
...
<message name="original">
  <part name="vehicle" type="xsd1:vehicleType"/>
  <part name="name" type="xsd:string"/>
</message>
<message name="transformed">
  <part name="vehicle" type="xsd:string"/>
  <part name="firstName" type="xsd:string"/>
  <part name="lastName" type="xsd:string"/>
</message>
...
<portType name="parkingLotMeter">
  <operation name="configure">
    <input name="oldClientInput" message="tns:original"/>
    <output name="updatedInput" message="tns:transformed"/>
  </operation>
...
</portType>
...
```

When the message is reconstructed, the transformer uses the input message's name, given in the input element, as the name of the root element of the XML document. It then uses the message parts and the schema types to recreate the data as an XML message. So if the transformer was using the contract defined in Example 86 an input message processed by the transformer could look like Example 87.

Example 87. Transformer Input Message

```
<ns1:oldClientInput xmlns:ns1="vehicle.demo.example">
  <vehicle>
<vin>0123456789</vin>
<model>Prius</model>
  </vehicle>
  <name>Old MacDonald</name>
</oldClientInput>
```

Outbound message

The results from the transformer go through the reverse of the process that turns the input message into a virtual XML document. The transformer uses the output message definition from the Artix contract to place the result message back onto the wire in the proper payload format. If the result message is not properly formed this attempt will fail, so you must be careful when writing your XSLT script to ensure that the results match the expected format.

When the result message is deconstructed, the transformer expects the following:

- If the output message is defined using the doc-literal style, the message must match the schema defining the message's part.
- If the output message is not defined using the doc-literal style, then the following must be true:
    - The name of the message's root element is the QName of the message element referred to by the `output` element in the contract.
    - There are the same number of elements in the result as there are `part` elements in the output message definition.
    - The elements in the result are based on the `name` attributes of the `part` elements in the output message definition.
    - The data contained in the element representing the output message's `part` elements matched the XML Schema definitions in the contract.
    - For example, a result message for the configure operation defined in Example 86 would look like Example 88.

Example 88. Transformer Output Message

```
<ns1:updatedInput xmlns:ns1="vehicle.demo.example">>
  <vehicle>Prius</vehicle>
  <firstName>Old</firstName>
  <lastName>MacDonald</lastName>
</updatedInput>
```

Using element names

You can configure the transformer to use the element name of the message parts instead of the value of the `part` element's `name` attribute. For more information see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

Writing XSLT Scripts

XML Stylesheet Language Transformations (XSLT) is a language used to describe the transformation of XML documents. The current W3C standard for XSLT is 1.0 and can be read at the W3C web site (http://www.w3.org/TR/xslt). XSLT documents, called scripts, are well-formed XML documents that describe how a source XML document is transformed into a resulting XML document. It can be used to perform tasks as simple as splitting a name entry into first and last name entries and as complex as validating that a complex XML document matches the expectations of an interface described in a WSDL document.

Procedure

Writing an XSLT script can be done in a number of ways and using a number of tools. The steps given here assume that you are writing fairly simple scripts using a text editor.

To write a XSLT script do the following:

1. Create an XML stylesheet with the required `xsl:transform` element.
2. Determine which elements in your source message need to be processed and create `xsd:template` elements for each of them.
3. For each element that has a matching template element, define how you want the element processed to produce a new output document.
4. If child elements need to be processed as part of processing a parent element, define a template for the child element and apply it as part of the parent element's template using `xsd:apply-templates`.

Elements of an XSLT Script

An XSLT script is essentially an XML stylesheet containing a special set of elements that instruct an XSLT engine in the processing of other XML documents. An XSLT script must be defined in an `xsl:transform` element or an `xsl:stylesheet` element. In addition, it needs at least one valid top-level element to define the transformation.

The transform element

The `xsl:transform` element denotes that the document is an XML stylesheet. The `xsl:stylesheet` element can be used in place of the `xsl:transform` element. They are equivalent.

When creating an XSLT script you must set the version attribute to 1.0 to inform the transformer what version of XSLT you are using. In addition, you must provide an XML namespace shortcut for the XSLT namespace in the `xsl:transform` element. Example 89 shows a valid `xsl:transform` element for an XSLT script.

Example 89. XSLT Script Stylesheet Element

```
<xsl:transform version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...
</stylesheet>
```

Top-level elements

While all that is needed to make an XML document a valid XSLT script is the `xsl:transform` element, the `xsl:transform` element does not provide any instructions for processing data. The data processing instructions in an XSLT script are provided by a number of top-level XSLT elements. These elements include:

- `xsl:import`
- `xsl:include`
- `xsl:strip-space`
- `xsl:preserve-space`
- `xsl:output`
- `xsl:key`
- `xsl:decimal-format`
- `xsl:namespace-alias`
- `xsl:attribute-set`
- `xsl:variable`
- `xsl:param`
- `xsl:template`

An XSLT script can have any number and combination of top-level elements. Other than `xsl:import`, which must occur before any other elements, the top-level elements can be used in any order. However, be aware that the order determines the order in which processing steps happen.

Example

Example 90 shows a simple XSLT script that transforms SSN elements into acctNum elements.

Example 90. Simple XSLT Script

```
<xsl:transform version = '1.0'
       xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="SSN">
<acctNum>
     <xsl:value-of select="."/>
</acctNum>
</xsl:template>
</xsl:stylesheet>
```

Using this XSLT script the transformer would change a message that contained `<SSN>012457890</SSN>` into a message that contained `<acctNum>012457890</acctNum>`.

XSLT Templates

XSLT processors use templates to determine the elements on which to apply a set of transformations. Documents are processed from the top element through their structure to determine if elements match a defined template. If a match is found, the rules specified by the template are applied.

To write a template in XSLT you need to do the following:

1. Create an `xsl:template` element.
2. Provide the path to the source element it processes.
3. Write the processing rules.

xsl:template elements

Templates are defined using `xsl:template` elements. These elements take one required attribute, `match`, which specifies the source element that triggers the rules. In addition, you can use the `name` attribute to give the template a unique identifier for referencing it elsewhere in the contract.

Specifying the source elements

You specify the elements of the source document to which template rules are matched using the `match` attribute of the `xsl:template` element. The source elements are specified using the syntax specified by the XPath specification (http://www.w3.org/TR/xpath). The source element address looks very similar to a file path where slash(/) specifies the root element and child elements are listed in top down order separated by a slash(/). For example to specify the `surname` element of the XML document shown in Example 91, you would specify it as `/name/surname`.

Example 91. Sample XML Document

```
<name>
  <firstname> Joe
  </firstname>
  <surname> Friday
  </surname>
<name>
```

Template matching order

XSLT processors start processing with the `<xsl:template match="/">` element if it is present. All of the processing directives for this template act on the top-level elements of the source document. For example, given the XML document shown in Example 91 any processing rules specified in `<xsl:template match="/">` would apply to the name element. In addition, specifying a template for the root element(/) forces you to make all your source element paths explicit from the root element. The XSLT script shown in Example 92 generates the string `Friday` when run on Example 91.

Example 92. XSLT Script with Root Element Template

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <xsl:value-of select="/name/surname"/>
  </xsl:template>
</xsl:transform>
```

You do not need to specify a template for the root element of the source document in an XSLT script. When you omit the root element’s template the processor treats all template paths as though they originated from the source documents top level element. The XSLT script in Example 93 generates the same output as the script in Example 92.

Example 93. XSLT Script without Root Element Template

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="surname">
       <xsl:value-of select="."/>
  </xsl:template>
</xsl:transform>
```

Template rules

The contents of an `xsl:template` element define how the source document is processed to produce an output document. You can use a combination of XSLT elements, HTML, and text to define the processing rules. Any plain text and HTML that are used in the processing rules are placed directly into the output document. For example, if you wanted to generate an HTML document from an XML document you would use an XSLT script that included HTML tags as part of its processing rules. The script in Example 94 takes an XML document with a `title` element and a `subTitle` element and produces an HTML document where the contents of `title` are displayed using the `<h1>` style and the contents of `subTitle` are displayed using the `<h2>` style.

Example 94. XSLT Template with HTML

```
<xsl:transform version = '1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <h1>
      <xsl:value-of select="//title"/>
    </h1>
    <h2>
      <xsl:value-of select="//subTitle"/>
    </h2>
  </xsl:template>
</xsl:transform>
```

Applying templates to child elements

You can instruct the XSLT processor to apply any templates defined in the script to the children of the element being processed using an `xsl:apply-templates` element as one of the rules in a template. `xsl:apply-templates` instructs the XSLT processor to treat the current element as a root element and run the templates in the script against it.

For example you could rewrite Example 94 as shown in Example 95 using `xsl:apply-templates` and defining a template for the `title` and `subTitle` elements.

Example 95. XSLT Template Using apply-templates

```
<xsl:transform version = '1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl"template match="title">
    <h1>
      <xsl:value-of select="."/>
    </h1>
  </xsl:template>
  <xsl"template match="subTitle">
    <h2>
      <xsl:value-of select="."/>
    </h2>
  </xsl:template>
</xsl:transform>
```

You can use the optional `select` attribute to limit the child elements to which the templates are applied. `select` takes an XPath value and operates in the same manner as the `match` attribute of `xsl:template`.

Example

For example, if your ordering system produced bills that looked similar to the XML document in, you could use an XSLT script to reformat the bill for a system that required the customer's name in a single element, name, and the city and state to be in a comma-separated field, `city`.

Example 96. Bill XML Document

```
<widgetBill>
<customer>
   <firstName> Joe
   </firstName>
   <lastName> Cool
   </lastName>
</customer>
<address>
   <street>
     123 Main Street
   </street>
   <city>
     Hot Coffee
   </city>
   <state> MS
   </state>
   <zipCode> 3942
   </zipCode>
</address>
  <amtDue> 123.50
</amtDue>
</widgetBill>
```

The XSLT script shown in Example 97 would result in the desired transformation.

Example 97. XSLT Script for widgetBill

```
<xsl:transform version = '1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
❶  <xsl:template match="widgetBill">
   <xsl:element name="widgetBill">
     <xsl:apply-templates/>
   </xsl:element>
  </xsl:template>
<xsl:template match="customer">
❷    <xsl:element name="name">
   <xsl:value-of select="concat(//firstName,' ',//lastName)"/>
   </xsl:element>
  </xsl:template>
  <xsl:template match="address">
```

```
❸   <xsl:element name="address">
     <xsl:copy-of select="//street"/>
     <xsl:element name="city">
       <xsl:value-of select="concat(//city,’, ’,//state)"/>
     </xsl:element>
     <xsl:copy-of select="//zipCode"/>
   </xsl:element>
  </xsl:template>
❹  <xsl:template match="amtDue">
<xsl:copy-of select="."/>
  </xsl:template>
</xsl:transform>
```

The script does the following:

❶ Creates an element, `widgetBill`, in the output document and places the results of the other templates as its children.

❷ Creates an element, name, and sets its value to the result of the concatenation.

❸ Creates an element, `address`, and sets its value to the results of the rules. `address` will contain a copy of the `street` element from the source document, a new element, `city`, that is a concatenation, and a copy of the `zipCode` element from the source document.

❹ Copies the `amtDue` element from the source document into the output document.

Processing the document in Example 96 with this XSLT script would result in the XML document shown in Example 98.

Example 98. Processed Bill XML Document

```
<widgetBill>
  <customer> Joe Cool
</customer>
<address>
<street>
     123 Main Street
   </street>
<city>
     Hot Coffee, MS
</city>
   <zipCode> 3942
   </zipCode>
</address>
  <amtDue> 123.50
</amtDue>
</widgetBill>
```

Common XSLT Functions

XSLT provides a range of capabilities in processing XML documents. These include conditional statements, looping, creating variables, and sorting.

However, there are a few common functions that are used to generate output documents. These include:

- xsl:value-of
- xsl:copy-of
- xsl:element

xsl:value-of

`xsl:value-of` creates a text node in the output document. It has a required `select` attribute that specifies the text to be inserted into the output document.

The value of `select` is evaluated as an expression describing the data to insert. It can contain any of the XSLT string functions, such as `concat()`, or an XSLT axis describing an element in the source document.

Once the `select` expression is evaluated the result is placed in the output document.

xsl:copy-of

`xsl:copy-of` copies data from the source document into the output document. It has a required `select`. The value of `select` is an expression describing the elements to be copied.

When the result of evaluating the expression is a tree fragment, the complete fragment is copied into the output document. When the result is an element, the element, its attributes, its namespaces, and its children are copied into the output document. When the result is neither an element nor a result tree fragment, the result is converted to a string and then inserted into the output document.

xsl:element

`xsl:element` creates an element in the output document. It takes a required `name` attribute that specifies the name of the element that is created. In addition, you can specify a `namespace` for the element using the optional namespace attribute.

Using Codeset Conversion

Some bindings do not natively support codeset conversion. Artix provides WSDL extensions and a plug-in that add codeset conversion to these bindings.

While many of the bindings supported by Artix provide a means for handling codeset conversion, some do not. It is also possible that any custom bindings you developed do not support codeset conversion. To allow bindings that do not natively support codeset conversion to participate in environments where more than one codeset is used, Artix provides an i18n message-level interceptor that will perform codeset conversion on the message buffer before it is placed on the wire.

The i18n interceptor can be configured by defining the codeset conversion in your endpoint's Artix contract using an Artix port extensor. You can also configure the i18n interceptor programmatically using the context mechanism. The programmatic settings will override any settings described in the contract. For more information on using the context mechanism see the appropriate development guide for your development environment.

Configuring Artix to use the i18n interceptor

Before your application can use the generic i18n interceptor for code conversion you must configure the Artix bus to load the required plug-ins and add the interceptor to the appropriate message interceptor lists. To configure your application to use the i18n interceptor do the following:

1. If your application includes a client that needs to use codeset conversion, add `i18n-context:I18nInterceptorFactory` to the binding:artix:client_message_interceptor_list variable for your application.

2. If your application includes a service that needs to use codeset conversion, add `i18n-context:I18nInterceptorFactory` to the binding:artix:server_message_interceptor_list variable for your application.

For more information on configuring Artix see ***Configuring and Deploying Artix Solutions, C++ Runtime***.

Describing the codeset conversions in the contract

You define the codeset conversions performed by the i18n interceptor in the `port` element defining an endpoint. There are two extensors used to define the codeset conversions. One, `i18n-context:server`, is for service providers and the other, `i18n-context:client`, is for clients. They both provide settings for how both incoming messages and outgoing messages are to be encoded. These extensions are defined in the namespace `http://schemas.iona.com/bus/i18n/context`.

To define the codeset conversions performed by the i18n interceptor do the following:

1. Add the following line to the `definitions` element of your contract.

```
xmlns:i18n-context="http://schemas.iona.com/bus/i18n/context"
```

2. If your application provides a service that requires codeset conversion add a `i18n-context:server` element to the port definition of the service endpoint.

Table 39 shows the attributes for the `i18n-context:server` element. These attributes define how message codesets are converted.

Table 39. Attributes for the `i18n-context:server` Element

Attribute	Description
`LocalCodeSet`	Specifies the server's native codeset. The default is the codeset specified by the local system's `locale` setting.
`OutboundCodeSet`	Specifies the codeset into which replies are converted. The default is the codeset specified in `InboundCodeSet`.
`InboundCodeSet`	Specifies the codeset into which requests are converted. The default is the codeset specified in `LocalCodeSet`.

If your application includes a client that requires codeset conversion add an `i18n-context:client` element to the port definition of the service endpoint.

Table 40 describes the attributes used by the `i18n-context:client` element for defining how message codesets are converted.

Table 40. Attributes for the `i18n-context:client` Element

Attribute	Description
`LocalCodeSet`	Specifies the server's native codeset. Default is the codeset specified by the local system's locale setting.
`OutboundCodeSet`	Specifies the codeset into which requests are converted. The default is the codeset specified in `LocalCodeSet`.
`InboundCodeSet`	Specifies the codeset into which replies are converted. The default is the codeset specified in `OutboundCodeSet`.

Example

The contract fragment in Example 99 shows a port definition for an endpoint that defines a server/client pair. The server uses UTF-8 as its local codeset and the client uses ISO-8859-1 as its local codeset.

Example 99. Specifying Codeset Conversion

```
...
<service name="covertedService">
  <port binding="tns:convertedFixedBinding"
       name="convertedPort">
    <http:address location="localhost:0"/>
    <i18n:client LocalCodeSet="ISO-8859-1"
                 OutboundCodeSet="UTF-8"
                 InboundCodeSet="ISO-8859-1"/>
    <i18n:server LocalCodeSet="UTF-8"
                 OutboundCodeSet="ISO-8859-1"/>
</port>
</service>
...
```

Using the endpoint definition above, the client will convert its requests into UTF-8 before sending them to the server. The server will convert its replies into ISO-8859-1 before sending them to the client. The client’s inbound codeset is set to ISO-8859-1 because if left unset the value would have defaulted to UTF-8. The client would then perform an extra conversion.