MICRO FOCUS

Artix 5.6.3

WSDL Extension Reference: C++

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

http://www.microfocus.com

2015-02-11

Contents

Part II Ports

Preface

What is Covered in this Book

This book is a reference to all of the Artix ESB specific WSDL extensions used in Artix contracts.

Who Should Read this Book

This book is intended for Artix users who are familiar with Artix concepts including:

- WSDL
- XMLSchema
- Artix interface design

In addition, this book assumes that the reader is familiar with the transports and middleware implementations with which they are working.

How to Use this Book

This book contains the following parts:

- Part I “Bindings”—contains descriptions for all the WSDL extensions used to define the payload formats supported by Artix.
- Part II “Ports”—contains descriptions for all the WSDL extensions used to define the transports supported by Artix.
- Part III “Other Extensions”—contains descriptions for the WSDL extensions used by Artix to support features like routing.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see ***Using the Artix Library***, available with the Artix documentation at https://supportline.microfocus.com/productdoc.aspx.

Contacting Micro Focus

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

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.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

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

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

Information We Need

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

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

Contact information

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

Additional technical information or advice is available from several sources.

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

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

You may want to check these URLs in particular:

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

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

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

Part I

Bindings

In this part

This part contains the following chapters:

SOAP 1.1 Binding

This chapter describes the extensions used to define a SOAP 1.1 message.

Runtime Compatibility

The SOAP binding is defined by a standard set of WSDL extensors.

soap:binding

Synopsis

```
<soap:binding style="..." transport="..." />
```

Description

The `soap:binding` element specifies that the payload format to use is a SOAP 1.1 message. It is a child of the WSDL `binding` element.

Attributes

The following attributes are defined within the `soap:binding` element.

- style
- transport

style

The value of the `style` attribute within the `soap:binding` element acts as the default for the `style` attribute within each `soap:operation` element. It indicates whether request/response operations within this binding are RPC-based (that is, messages contain parameters and return values) or document-based (that is, messages contain one or more documents).

Valid values are `rpc` and `document`. The specified value determines how the SOAP `Body` element within a SOAP message is structured.

If `rpc` is specified, each message part within the SOAP `Body` element is a parameter or return value and will appear inside a wrapper element within the SOAP `Body` element. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:body namespace` attribute. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. Each part name must match the parameter name to which it corresponds.

For example, the SOAP `Body` element of a SOAP request message is as follows if the style is RPC-based:

```
<SOAP-ENV:Body>
    <m:GetStudentGrade xmlns:m="URL">
        <StudentCode>815637</StudentCode>
        <Subject>History</Subject>
    </m:GetStudentGrade>
</SOAP-ENV:Envelope>
```

If `document` is specified, message parts within the SOAP `Body` element appear directly under the SOAP `Body` element as body entries and do not appear inside a wrapper element that

corresponds to an operation. For example, the SOAP `Body` element of a SOAP request message is as follows if the style is document-based:

```
<SOAP-ENV:Body>
    <StudentCode>815637</StudentCode>
    <Subject>History</Subject>
</SOAP-ENV:Envelope>
```

transport

The `transport` attribute defaults to the URL that corresponds to the HTTP binding in the W3C SOAP specification (http://schemas.xmlsoap.org/soap/http). If you want to use another transport (for example, SMTP), modify this value as appropriate for the transport you want to use.

soap:operation

Synopsis

```
<soap:operation style="..." soapAction="..." />
```

Description

The `soap:operation` element is a child of the WSDL `operation` element. A `soap:operation` element is used to encompass information for an operation as a whole, in terms of input criteria, output criteria, and fault information.

Attributes

The following attributes are defined within a `soap:operation` element:

- style
- soapAction

style

This indicates whether the relevant operation is RPC-based (that is, messages contain parameters and return values) or document-based (that is, messages contain one or more documents).

Valid values are `rpc` and `document`. The default value for `soap:operation style` is based on the value specified for the `soap:binding style` attribute.

See "style" on page 3 for more details of the `style` attribute.

soapAction

This specifies the value of the `SOAPAction` HTTP header field for the relevant operation. The value must take the form of the absolute URI that is to be used to specify the intent of the SOAP message.

Note: This attribute is mandatory only if you want to use SOAP over HTTP. Leave it blank if you want to use SOAP over any other transport.

soap:body

Synopsis

```
<soap:body use="..." encodingStyle="..." namespace="..."
parts="..." />
```

Description

The `soap:body` element in a binding is a child of the `input`, `output`, and `fault` child elements of the WSDL `operation` element. A `soap:body` element is used to provide information on how message

parts are to be appear inside the body of a SOAP message. As explained in "soap:operation" on page 4, the structure of the SOAP `Body` element within a SOAP message is dependent on the setting of the `soap:operation style` attribute.

Attributes

The following attributes are defined within a `soap:body` element:

- use
- encodingStyle
- namespace
- parts

use

This mandatory attribute indicates how message parts are used to denote data types. Each message part relates to a particular data type that in turn might relate to an abstract type definition or a concrete schema definition.

An abstract type definition is a type that is defined in some remote encoding schema whose location is referenced in the WSDL contract via an `encodingStyle` attribute. In this case, types are serialized based on the set of rules defined by the specified encoding style.

A concrete schema definition relates to types that are defined in the WSDL contract itself, within a `schema` element within the `types` component of the contract.

The following are valid values for the `use` attribute:

- `encoded`
- `literal`

If `encoded` is specified, the `type` attribute that is specified for each message part (within the `message` component of the WSDL contract) is used to reference an abstract type defined in some remote encoding schema. In this case, a concrete SOAP message is produced by applying encoding rules to the abstract types. The encoding rules are based on the encoding style identified in the `soap:body encodingStyle` attribute. The encoding takes as input the `name` and `type` attribute for each message part (defined in the `message` component of the WSDL contract). If the encoding style allows variation in the message format for a given set of abstract types, the receiver of the message must ensure they can understand all the format variations.

If `literal` is specified, either the `element` or `type` attribute that is specified for each message part (within the `message` component of the WSDL contract) is used to reference a concrete schema definition (defined within the `types` component of the WSDL contract). If the `element` attribute is used to reference a concrete schema definition, the referenced element in the SOAP message appears directly under the SOAP `Body` element (if the operation style is document-based) or under a part accessor element that has the same name as the message part (if the operation style is RPC-based). If the `type` attribute is used to reference a concrete schema definition, the referenced type in the SOAP message becomes the schema type of the SOAP `Body` element (if the operation style is documented-based) or of the part accessor element (if the operation style is document-based).

encodingStyle

This attribute is used when the `soap:body use` attribute is set to `encoded`. It specifies a list of URIs (each separated by a space) that represent encoding styles that are to be used within the SOAP message. The URIs should be listed in order, from the most restrictive encoding to the least restrictive.

This attribute can also be used when the `soap:body use` attribute is set to `literal`, to indicate that a particular encoding was used to derive the concrete format, but that only the specified variation is supported. In this case, the sender of the SOAP message must conform exactly to the specified schema.

namespace

If the `soap:operation style` attribute is set to `rpc`, each message part within the SOAP `Body` element of a SOAP message is a parameter or return value and will appear inside a wrapper element within the SOAP `Body` element. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:body namespace` attribute.

parts

This attribute is a space separated list of parts from the parent `input`, `output`, or `fault` element. When parts is set, only the specified parts of the message are included in the SOAP `Body` element. The unlisted parts are not transmitted unless they are placed into the SOAP header.

soap:header

Synopsis

```
<soap:header message="..." part="..." use="..."
encodingStyle="..." namespace="..."/>
```

Description

The `soap:header` element in a binding is an optional child of the `input`, `output`, and `fault` elements of the WSDL `operation` element. A `soap:header` element defines the information that is placed in a SOAP header element. You can define any number of `soap:header` elements for an operation. As explained in “soap:operation” on page 4, the structure of the SOAP header within a SOAP message is dependent on the setting of the `soap:operation` element’s `style` attribute.

Attributes

The `soap:header` element has the following attributes.

`message`	Specifies the qualified name of the message from which the contents of the SOAP header is taken.
`part`	Specifies the name of the message part that is placed into the SOAP header.
`use`	Used in the same way as the `use` attribute within the `soap:body` element. See "use" on page 5 for more details.
`encodingStyle`	Used in the same way as the `encodingStyle` attribute within the `soap:body` element. See "encodingStyle" on page 6 for more details.
`namespace`	If the `soap:operation style` attribute is set to `rpc`, each message part within the SOAP header of a SOAP message is a parameter or return value and will appear inside a wrapper element within the SOAP header. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:header namespace` attribute.

soap:fault

Synopsis

```
<soap:fault name="..." use="..." encodingStyle="..." />
```

Description

The `soap:fault` element is a child of the WSDL `fault` element within an `operation` component. Only one `soap:fault` element is defined for a particular operation. The operation must be a request-response or solicit-response type of operation, with both `input` and `output` elements. The `soap:fault` element is used to transmit error and status information within a SOAP response message.

Note: A fault message must consist of only a single message part. Also, it is assumed that the `soap:operation` element's `style` attribute is set to `document`, because faults do not contain parameters.

Attributes

The `soap:fault` element has the following attributes:

`name`	Specifies the name of the fault. This relates back to the `name` attribute for the `fault` element specified for the corresponding operation within the `portType` component of the WSDL contract.
`use`	This attribute is used in the same way as the `use` attribute within the `soap:body` element. See "use" on page 5 for more details.
`encodingStyle`	This attribute is used in the same way as the `encodingStyle` attribute within the `soap:body` element. See "encodingStyle" on page 6 for more details.

SOAP 1.2 Binding

This chapter describes the extensions used to define a SOAP 1.2 message.

Runtime Compatibility

The SOAP 1.2 binding is defined by a standard set of WSDL extensors.

wsoap12:binding

Synopsis

```
<wsoap12:binding style="..." transport="..." />
```

Description

The `wsoap12:binding` element specifies that the payload format to use is a SOAP 1.2 message. It is a child of the WSDL `binding` element.

Attributes

The following attributes are defined within the `wsoap12:binding` element.

- style
- transport

style

The value of the `style` attribute acts as the default for the `style` attribute within each wsoap12:operation element. It indicates whether request/response operations within this binding are RPC-based (that is, messages contain parameters and return values) or document-based (that is, messages contain one or more documents).

Valid values are `rpc` and `document`. The specified value determines how the SOAP `Body` element within a SOAP message is structured.

If `rpc` is specified, each message part within the SOAP `Body` element is a parameter or return value and will appear inside a wrapper element within the SOAP `Body` element. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:body namespace` attribute. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. Each part name must match the parameter name to which it corresponds.

For example, the SOAP `Body` element of a SOAP request message is as follows if the style is RPC-based:

```
<SOAP-ENV:Body>
    <m:GetStudentGrade xmlns:m="URL">
        <StudentCode>815637</StudentCode>
        <Subject>History</Subject>
    </m:GetStudentGrade>
</SOAP-ENV:Envelope>
```

If `document` is specified, message parts within the SOAP `Body` element appear directly under the SOAP `Body` element as body entries and do not appear inside a wrapper element that

corresponds to an operation. For example, the SOAP `Body` element of a SOAP request message is as follows if the style is document-based:

```
<SOAP-ENV:Body>
    <StudentCode>815637</StudentCode>
    <Subject>History</Subject>
</SOAP-ENV:Envelope>
```

transport

The `transport` attribute specifies a URL describing the SOAP transport to which this binding corresponds. The URL that corresponds to the HTTP binding in the W3C SOAP specification is `http://schemas.xmlsoap.org/soap/http`. If you want to use another transport (for example, SMTP), modify this value as appropriate for the transport you want to use.

wsoap12:operation

Synopsis

```
<wsoap12:operation style="..." soapAction="..."
soapActionRequired="..."/>
```

Description

The `wsoap12:operation` element is a child of the WSDL `operation` element. A `soap:operation` element is used to encompass information for an operation as a whole, in terms of input criteria, output criteria, and fault information.

Attributes

The following attributes are defined within a `wsoap12:operation` element:

- style
- soapAction
- soapActionRequired

style

This indicates whether the relevant operation is RPC-based (that is, messages contain parameters and return values) or document-based (that is, messages contain one or more documents).

Valid values are `rpc` and `document`. The default value for the `wsoap12:operation` element's `style` attribute is based on the value specified for the wsoap12:binding element's `style` attribute.

soapAction

This specifies the value of the `SOAPAction` HTTP header field for the relevant operation. The value must take the form of the absolute URI that is to be used to specify the intent of the SOAP message.

Note: This attribute is mandatory only if you want to use SOAP 1.2 over HTTP. Leave it blank if you want to use SOAP 1.2 over any other transport.

soapActionRequired

The `soapActionRequired` is a boolean that specifies if the value of the soapAction attribute must be conveyed in the request message. When the value of `soapActionRequired` is `true`, the soapAction attribute must be present. The default is to `true`.

wsoap12:body

Synopsis

```
<wsoap12:body use="..." encodingStyle="..." namespace="..."
parts="..." />
```

Description

The `wsoap12:body` element in a binding is a child of the `input`, `output`, and `fault` child elements of the WSDL `operation` element. A `wsoap12:body` element is used to provide information on how message parts are to be appear inside the body of a SOAP 1.2 message. As explained in "wsoap12:operation" on page 10, the structure of the SOAP `Body` element within a SOAP message is dependent on the setting of the `soap:operation style` attribute.

Attributes

The following attributes are defined within a `wsoap12:body` element:

- use
- encodingStyle
- namespace
- parts

use

This mandatory attribute indicates how message parts are used to denote data types. Each message part relates to a particular data type that in turn might relate to an abstract type definition or a concrete schema definition.

An abstract type definition is a type that is defined in some remote encoding schema whose location is referenced in the WSDL contract via an `encodingStyle` attribute. In this case, types are serialized based on the set of rules defined by the specified encoding style.

A concrete schema definition relates to types that are defined in the WSDL contract itself, within a `schema` element within the `types` component of the contract.

The following are valid values for the `use` attribute:

- `literal`
- `encoded`

Note: Artix does not support encoded messages when using SOAP 1.2.

If `literal` is specified, either the `element` or `type` attribute that is specified for each message part (within the `message` component of the WSDL contract) is used to reference a concrete schema definition (defined within the `types` component of the WSDL contract). If the `element` attribute is used to reference a concrete schema definition, the referenced element in the SOAP 1.2 message appears directly under the SOAP `Body` element (if the operation style is document-based) or under a part accessor element that has the same name as the message part (if the operation style is RPC-based). If the `type` attribute is used to reference a concrete schema definition, the referenced type in the SOAP 1.2 message becomes the schema type of the SOAP `Body` element (if the operation style is documented-based) or of the part accessor element (if the operation style is document-based).

encodingStyle

This attribute is only used when the `wsoap12:body` element's `use` attribute is set to `encoded`. and the `wsoap12:binding` element's `style` attribute is set to `rpc`. It specifies the URI that represents the encoding rules that used to construct the SOAP 1.2 message.

namespace

If the `soap:operation` element's `style` attribute is set to `rpc`, each message part within the SOAP `Body` element of a SOAP 1.2 message is a parameter or return value and will appear inside a wrapper element within the SOAP `Body` element. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:body` `namespace` attribute.

parts

This attribute is a space separated list of parts from the parent `input`, `output`, or `fault` element. When the `parts` attribute is set, only the specified parts of the message are included in the SOAP `Body` element. The unlisted parts are not transmitted unless they are placed into the SOAP header.

wsoap12:header

Synopsis

```
<wsoap12:header message="..." part="..." use="..."
encodingStyle="..." namespace="..."/>
```

Description

The `wsoap12:header` element in a binding is an optional child of the `input`, `output`, and `fault` elements of the WSDL `operation` element. A `wsoap12:header` element defines the information that is placed in a SOAP 1.2 header element. You can define any number of `wsoap12:header` elements for an operation. As explained in "wsoap12:operation" on page 10, the structure of the header within a SOAP 1.2 message is dependent on the setting of the `wsoap12:operation` element's `style` attribute.

Attributes

The `wsoap12:header` element has the following attributes.

`message`	Specifies the qualified name of the message from which the contents of the SOAP header is taken.
`part`	Specifies the name of the message part that is placed into the SOAP header.
`use`	Used in the same way as the `wsoap12:body` element's `use` attribute.
`encodingStyle`	Used in the same way as the `wsoap12:body` element's `encodingStyle` attribute.
`namespace`	Specifies the namespace to be assigned to the header element when the `use` attribute is set to `encoded`. The header is constructed in all cases as if the `wsoap12:binding` element's `style` attribute had a value of `document`.

wsoap12:fault

Synopsis

```
<wsoap12:fault name="..." namespace="..." use="..."
encodingStyle="..." />
```

Description

The `wsoap12:fault` element is a child of the WSDL `fault` element within a WSDL `operation` element. The operation must have both `input` and `output` elements. The `wsoap12:fault` element is used to transmit error details and status information within a SOAP 1.2 response message.

Note: A fault message must consist of only a single message part. Also, it is assumed that the wsoap12:operation element's `style` attribute is set to `document`, because faults do not contain parameters.

Attributes

The `wsoap12:fault` element has the following attributes:

`name`	Specifies the name of the fault. This relates back to the `name` attribute for the `fault` element specified for the corresponding operation within the `portType` component of the WSDL contract.
`namespace`	Specifies the namespace to be assigned to the wrapper element for the fault. This attribute is ignored if the `style` attribute of either the wsoap12:binding element of the containing binding or of the wsoap12:operation element of the containing operation is either omitted or has a value of `document`. This attribute is required if the value of the wsoap12:binding element's `style` attribute is set to `rpc`.
`use`	This attribute is used in the same way as the wsoap12:body element's `use` attribute.
`encodingStyle`	This attribute is used in the same way as the wsoap12:body element's `encodingStyle` attribute

MIME Multipart/Related Binding

This chapter describes the extensions that are used to define a SOAP message binding that contains binary data.

Runtime Compatibility

The MIME extensions are defined by a standard.

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 `defintion` element to set this up is shown in Example 1.

Example 1: *MIME Namespace Specification in a Contract*

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

mime:multipartRelated

Synopsis

```
<mime:multipartRelated>
  <mime:part ...>
    ...
  </mime:part>
  ...
</mime:multipartRelated>
```

Description

The `mime:multipartRelated` element is the child of an `input` element or an `output` element that is part of a SOAP binding. It tells Artix that the message body is going to be a multipart message that potentially contains binary data. `mime:multipartReleated` elements in Artix contain one or more mime:part elements that describe the individual parts of the message.

mime:part

Synopsis

```
<mime:part name="...">
  ...
</mime:part>
```

Description

The `mime:part` element is the child of a mime:multipartRelated element. It is used to define the parts of a multi-part message. The first `mime:part` element must contain the soap:body element or the wsoap12:body element that would normally appear in a SOAP binding. The remaining `mime:part` elements define the attachments that are being sent in the message using a mime:content element.

Attributes

The `mime:part` element has a single attribute called `name`. `name` is a unique string that is used to identify the part being described.

mime:content

Synopsis

```
<mime:content part="..." type="..." />
```

Description

The `mime:content` element is the child of a mime:part element. It defines the binary content being passed as an attachment to a SOAP message.

Attributes

The `mime:content` element has the following attributes:

`part` — Specifies the name of the WSDL `part` element, 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` — Specifies 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 IANA and described in the following:

- *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* (https://www.ietf.org/rfc/rfc2045.txt)
- *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* (https://www.ietf.org/rfc/rfc2046.txt).

CORBA Binding and Type Map

Artix CORBA support uses a combination of a WSDL binding element and a corba:typeMapping element to unambiguously define CORBA Messages.

This chapter discusses the following topics:

CORBA Binding Extension Elements

Runtime Compatibility

The CORBA binding extensions are compatible with the C++ runtime.

C++ Runtime Namespace

The WSDL extensions used for the C++ Runtime CORBA binding and the CORBA data mappings are defined in the namespace `http://schemas.iona.com/bindings/corba`. The Artix designer adds the following namespace declaration to any contract that uses the C++ runtime CORBA binding:

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

Primitive Type Mapping

Most primitive IDL types are directly mapped to primitive XML Schema types. Table 1 lists the mappings for the supported IDL primitive types.

Table 1: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
Any	xsd:anyType	corba:any	IT_Bus::AnyHolder
boolean	xsd:boolean	corba:boolean	IT_Bus::Boolean
char	xsd:byte	corba:char	IT_Bus::Char
wchar	xsd:string	corba:wchar	
double	xsd:double	corba:double	IT_Bus::Double
float	xsd:float	corba:float	IT_Bus::Float
octet	xsd:unsignedByte	corba:octet	IT_Bus::Octet
long	xsd:int	corba:long	IT_Bus::Long
long long	xsd:long	corba:longlong	IT_Bus::LongLong

Table 1: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
short	xsd:short	corba:short	IT_Bus::Short
string	xsd:string	corba:string	IT_Bus::String
wstring	xsd:string	corba:wstring	
unsigned short	xsd:unsignedShort	corba:ushort	IT_Bus::UShort
unsigned long	xsd:unsignedInt	corba:ulong	IT_Bus::ULong
unsigned long long	xsd:unsignedLong	corba:ulonglong	IT_Bus::ULongLong
Object	wsa:EndpointReferenceType	corba:object	WS_Addressing::EndpointReferenceType
TimeBase::UtcT	xsd:dateTime[a]	corba:dateTime	IT_Bus::DateTime

a. The mapping between xsd:dateTime and TimeBase:UtcT is only partial. For the restrictions see "Unsupported time/date values" on page 18

Unsupported types

The following CORBA types are not supported:

- long double
- Value types
- Boxed values
- Local interfaces
- Abstract interfaces
- Forward-declared interfaces

Unsupported time/date values

The following xsd:dateTime values cannot be mapped to TimeBase::UtcT:

- Values with a local time zone. Local time is treated as a 0 UTC time zone offset.
- Values prior to 15 October 1582.
- Values greater than approximately 30,000 A.D.

The following TimeBase::UtcT values cannot be mapped to xsd:dateTime:

- Values with a non-zero inacclo or inacchi.
- Values with a time zone offset that is not divisible by 30 minutes.
- Values with time zone offsets greater than 14:30 or less than -14:30.
- Values with greater than millisecond accuracy.
- Values with years greater than 9999.

corba:binding

Synopsis

```
<corba:binding repositoryID="..." bases=".." />
```

Description

The `corba:binding` element indicates that the binding is a CORBA binding.

Attributes

This element has two attributes:

`repositoryID` A required attribute whose value is the full type ID of the CORBA interface. The type ID is embedded in an object's IOR and must conform to the format `IDL:`*`module`*`/`*`interface`*`:1.0`.

`bases` An optional attribute whose value is the type ID of the interface from which the interface being bound inherits.

Examples

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

Synopsis

```
<corba:operation name="..." >
  <corba:param ... />
  ...
  <corba:return ... />
  <corba:raises ... />
</corba:operation>
```

Description

The `corba:operation` element is a child element of the WSDL `operation` element and describes the parts of the operation's messages. It has one or more of the following children:

- corba:param
- corba:return
- corba:raises

Attributes

The `corba:operation` attribute takes a single attribute, `name`, which duplicates the name given in `operation`.

corba:param

Synopsis

```
<corba:param name="..." mode="..." idltype="..." />
```

Description

The `corba:param` element 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.

Attributes

The `corba:param` element has the following required attributes:

`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. See "Type Map Extension Elements" on page 21.
`name`	Specifies the name of the parameter as given in the `name` attribute of the corresponding `part` element.

corba:return

Synopsis

```
<corba:return name="..." idltype="..." />
```

Description

The `corba:return` element is a child of `corba:operation` and specifies the return type, if any, of the operation.

Attributes

The `corba:return` element has two attributes:

`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

Synopsis

```
<corba:raises exception="..." />
```

Description

The `corba:raises` element 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.

Attributes

The `corba:raises` element has one required attribute, `exception`, which specifies the type of data returned in the exception.

Type Map Extension Elements

corba:typeMapping

Synopsis

```
<corba:typeMapping
targetNamespace="http://schemas.iona.com/bindings/corba/typemap"
>
...
</corba:typeMapping>
```

Description

Because complex types (such as structures, arrays, and exceptions) require a more involved mapping to resolve type ambiguity, the full mapping for a complex type is described in a `corba:typeMapping` element in an Artix contract. This element contains a type map describing the metadata required to fully describe a complex type as a CORBA data type. This metadata may include the members of a structure, the bounds of an array, or the legal values of an enumeration.

Attributes

The `corba:typeMapping` element requires a `targetNamespace` attribute that specifies the namespace for the elements defined by the type map.

Examples

Table 2 shows the mappings from complex IDL types to Artix CORBA types.

Table 2: *Complex IDL Type Mappings*

IDL Type	CORBA Binding Type
struct	corba:struct
enum	corba:enum
fixed	corba:fixed
union	corba:union
typedef	corba:alias
array	corba:array
sequence	corba:sequence
exception	corba:exception

corba:struct

Synopsis

```
<corba:struct name="..." type="..." repositoryID="..." />
  <corba:member ... />
  ...
</corba:struct>
```

The `corba:struct` element is used to represent XMLSchema types that are defined using `complexType` elements. The elements of the structure are described by a series of corba:member elements.

Attributes

A `corba:struct` element requires three attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`repositoryID`	The fully specified repository ID for the CORBA type.

corba:member

Synopsis

```
<corba:member name="..." idlType="..." />
```

Description

The `corba:member` element is used to define the parts of the structure represented by the parent element. The elements must be declared in the same order used in the IDL representation of the CORBA type.

Attributes

A `corba:member` requires two attributes:

`name`	The name of the element
`idltype`	The IDL type of the element. This type can be either a primitive type or another complex type that is defined in the type map.

Examples

For example, you may have a structure, `personalInfo`, similar to the one in Example 2.

Example 2: *personalInfo*

```
enum hairColorType {red, brunette, blonde};

struct personalInfo
{
 string name;
 int age;
 hairColorType hairColor;
}
```

It can be represented in the CORBA type map as shown in Example 3.

Example 3: *CORBA Type Map for personalInfo*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:struct name="personalInfo" type="xsd1:personalInfo"
   repositoryID="IDL:personalInfo:1.0">
    <corba:member name="name" idltype="corba:string"/>
    <corba:member name="age" idltype="corba:long"/>
    <corba:member name="hairColor" idltype="corbatm:hairColorType"/>
  </corba:struct>
</corba:typeMapping>
```

The idltype `corbatm:hairColorType` refers to a complex type that is defined earlier in the CORBA type map.

corba:enum

Synopsis

```
<corba:enum name="..." type="..." repositoryID="...">
  <corba:enumerator ... />
  ...
</corba:enum>
```

The `corba:enum` element is used to represent enumerations. The values for the enumeration are described by a series of corba:enumerator elements.

Attributes

A `corba:enum` element requires three attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`repositoryID`	The fully specified repository ID for the CORBA type.

corba:enumerator

Synopsis

```
<corba:enumerator value="..." />
```

Description

The `corba:enumerator` element represents the values of an enumeration. The values must be listed in the same order used in the IDL that defines the CORBA enumeration.

Attributes

A `corba:enumerator` element takes one attribute, `value`.

Examples

For example, the enumeration defined in Example 2 on page 22, `hairColorType`, can be represented in the CORBA type map as shown in Example 4:

Example 4: *CORBA Type Map for hairColorType*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:enum name="hairColorType" type="xsd1:hairColorType"
   repositoryID="IDL:hairColorType:1.0">
    <corba:enumerator value="red"/>
    <corba:enumerator value="brunette"/>
    <corba:enumerator value="blonde"/>
  </corba:enum>
</corba:typeMapping>
```

corba:fixed

Synopsis

```
<corba:fixed name="..." repositoryID="..." type="..."
digits="..." scale="..." />
```

Description

Fixed point data types are a special case in the Artix contract mapping. A CORBA fixed type is represented in the logical portion of the contract as the XML Schema primitive type `xsd:decimal`. However, because a CORBA fixed type requires additional information to be fully mapped to a physical CORBA data type, it must also be described in the CORBA type map section of an Artix contract using a `corba:fixed` element.

Attributes

A `corba:fixed` element requires five attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`repositoryID`	The fully specified repository ID for the CORBA type.
`type`	The logical type the structure is mapping (for CORBA fixed types, this is always `xsd:decimal`).
`digits`	The upper limit for the total number of digits allowed. This corresponds to the first number in the fixed type definition.
`scale`	The number of digits allowed after the decimal point. This corresponds to the second number in the fixed type definition.

Examples

For example, the fixed type defined in Example 5, `myFixed`, would

Example 5: *myFixed Fixed Type*

```
\\IDL
typedef fixed<4,2> myFixed;
```

be described by a type entry in the logical type description of the contract, as shown in Example 6.

Example 6: *Logical description from myFixed*

```
<xsd:element name="myFixed" type="xsd:decimal"/>
```

In the CORBA type map portion of the contract, it would be described by an entry similar to Example 7. Notice that the description in the CORBA type map includes the information needed to fully represent the characteristics of this particular fixed data type.

Example 7: *CORBA Type Map for myFixed*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:fixed name="myFixed" repositoryID="IDL:myFixed:1.0" type="xsd:decimal"
   digits="4" scale="2"/>
</corba:typeMapping>
```

corba:union

Synopsis

```
<corba:union name="..." type="..." discriminator="..."
             repositoryID="...">
  <corba:unionbranch ... />
  ...
</corba:union>
```

Description

The `corba:union` element is used to resolve the relationship between a union's discriminator and its members. A corba:union element is required for every CORBA union defined in an IDL contract. The members of the union are described using a series of nested corba:unionbranch elements.

Attributes

A `corba:union` element has four mandatory attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`discriminator`	The IDL type used as the discriminator for the union.
`repositoryID`	The fully specified repository ID for the CORBA type.

corba:unionbranch

Synopsis

```
<corba:unionbranch name="..." idltype="..." default="...">
  <corba:case ... />
  ...
</corba:unionbranch>
```

Description

The `corba:unionbranch` element defines the members of a union. Each `corba:unionbranch` except for one describing the union's default member will have at least one corba:case element as a child.

Attributes

A `corba:unionbranch` element has two required attributes and one optional attribute.

`name`	A unique identifier used to reference the union member.
`idltype`	The IDL type of the union member. This type can be either a primitive type or another complex type that is defined in the type map.
`default`	The optional attribute specifying if this member is the default case for the union. To specify that the value is the default set this attribute to `true`.

corba:case

Synopsis

```
<corba:case label="..." />
```

Description

The `corba:case` element defines the explicit relationship between the discriminator's value and the associated union member.

Attributes

The `corba:case` element's only attribute, `label`, specifies the value used to select the union member described by the corba:unionbranch.

Examples

For example consider the union, `myUnion`, shown in Example 8:

Example 8: *myUnion IDL*

```
//IDL
union myUnion switch (short)
{
  case 0:
    string case0;
  case 1:
  case 2:
    float case12;
  default:
    long caseDef;
};
```

For example `myUnion`, Example 8, would be described with a CORBA type map entry similar to that shown in Example 9.

Example 9: *myUnion CORBA type map*

```
<corba:typeMapping
   targetNamespace="http://schemas.iona.com/bindings/corba/t
   ypemap">
...
  <corba:union name="myUnion" type="xsd1:myUnion"
   discriminator="corba:short"
   repositoryID="IDL:myUnion:1.0">
    <corba:unionbranch name="case0" idltype="corba:string">
      <corba:case label="0"/>
    </corba:unionbranch>
    <corba:unionbranch name="case12" idltype="corba:float">
      <corba:case label="1"/>
      <corba:case label="2"/>
    </corba:unionbranch>
    <corba:unionbranch name="caseDef" idltype="corba:long"
   default="true"/>
  </corba:union>
</corba:typeMapping>
```

corba:alias

Synopsis

```
<corba:alias name="..." type="..." repositoryID="..." />
```

Description

The `corba:alias` element is used to represent a `typedef` statement in an IDL contract.

Attributes

The `corba:alias` element has three attributes:

`name`	The value of the `name` attribute from the XMLSchema `simpleType` element representing the renamed type.
`type`	The XMLSchema type for the base type.
`repositoryID`	The fully specified repository ID for the CORBA type.

Examples

For example, the definition of myLong in Example 10, can be

Example 10: *myLong IDL*

```
//IDL
typedef long myLong;
```

described as shown in Example 11:

Example 11: *myLong WSDL*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="typedef.idl" ...>
  <types>
  ...
    <xsd:simpleType name="myLong">
      <xsd:restriction base="xsd:int"/>
    </xsd:simpleType>
  ...
  </types>
...
  <corba:typeMapping
   targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
    <corba:alias name="myLong" type="xsd:int"
   repositoryID="IDL:myLong:1.0" basetype="corba:long"/>
  </corba:typeMapping>
</definitions>
```

corba:array

Synopsis

```
<corba:array name="..." repositoryID="..." type="..."
elemtype="..." bound="..." />
```

Description

In the CORBA type map, arrays are described using a `corba:array` element.

Attributes

A `corba:array` has the following required attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`repositoryID`	The fully specified repository ID for the CORBA type.
`type`	The logical type the structure is mapping.
`elemtype`	The IDL type of the array's element. This type can be either a primitive type or another complex type that is defined within the type map.
`bound`	The size of the array.

Examples

For example, consider an array, `myArray`, as defined in Example 12.

Example 12: *myArray IDL*

```
//IDL
typedef long myArray[10];
```

The array `myArray` will have a CORBA type map description similar to the one shown in Example 13.

Example 13: *myArray CORBA type map*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
  <corba:array name="myArray" repositoryID="IDL:myArray:1.0" type="xsd1:myArray"
   elemtype="corba:long" bound="10"/>
</corba:typeMapping>
```

corba:sequence

Synopsis

```
<corba:sequence name="..." repositoryID="..." elemtype="..."
bound="..." />
```

Description

The `corba:sequence` element represents an IDL `sequence`.

Attributes

A `corba:sequence` has five required attributes.

`name`	A unique identifier used to reference the CORBA type in the binding.
`repositoryID`	The fully specified repository ID for the CORBA type.
`type`	The logical type the structure is mapping.
`elemtype`	The IDL type of the sequence's elements. This type can be either a primitive type or another complex type that is defined within the type map.
`bound`	The size of the sequence.

Examples

For example, consider the two sequences defined in Example 14, `longSeq` and `charSeq`.

Example 14: *IDL Sequences*

```
\\ IDL
typedef sequence<long> longSeq;
typedef sequence<char, 10> charSeq;
```

The sequences described in Example 14 has a CORBA type map description similar to that shown in Example 15.

Example 15: *CORBA type map for Sequences*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
    <corba:sequence name="longSeq" repositoryID="IDL:longSeq:1.0" type="xsd1:longSeq"
   elemtype="corba:long" bound="0"/>
    <corba:sequence name="charSeq" repositoryID="IDL:charSeq:1.0" type="xsd1:charSeq"
   elemtype="corba:char" bound="10"/>
  </corba:typeMapping>
```

corba:exception

Synopsis

```
<corba:exception name="..." type="..." repositoryID="...">
  <corba:member ... />
  ...
</corba:exception>
```

Description

The `corba:exception` element is a child of a corba:typeMapping element. It describes an exception in the CORBA type map. The pieces of data returned with the exception are described by a series of corba:member elements. The elements must be declared in the same order as in the IDL representation of the exception.

Attributes

A `corba:exception` element has the following required attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`repositoryID`	The fully specified repository ID for the CORBA type.

Examples

For example, consider the exception `idNotFound` defined in Example 16.

Example 16: *idNotFound Exception*

```
\\IDL
exception idNotFound
{
  short id;
};
```

In the CORBA type map portion of the contract, `idNotFound` is described by an entry similar to that shown in Example 17:

Example 17: *CORBA Type Map for idNotFound*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:exception name="idNotFound" type="xsd1:idNotFound"
   repositoryID="IDL:idNotFound:1.0">
    <corba:member name="id" idltype="corba:short"/>
  </corba:exception>
</corba:typeMapping>
```

corba:anonsequence

Synopsis

```
<corba:anonsequence name="..." bound="..." elemtype="..."
type="..." />
```

Description

The `corba:anonsequence` element is used when representing recursive types. Because XMLSchema recursion requires the use of two defined types and IDL recursion does not, the CORBA type map uses the `corba:anonsequence` element as a means of bridging the gap. When Artix generates IDL from a contract, it will not generate new IDL types for XMLSchema types that are used in a `corba:anonsequence` element.

Attributes

The `corba:anonsequence` element has four required attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`bound`	The size of the sequence.
`elemtype`	The name of the CORBA type map element that defines the contents of the sequence.

`type` The logical type the element represents.

Examples

Example 18 shows a recursive XMLSchema type, `allAboutMe`, defined using a named type.

Example 18: *Recursive XML Schema Type*

```
<complexType name="allAboutMe">
  <sequence>
    <element name="shoeSize" type="xsd:int"/>
    <element name="mated" type="xsd:boolean"/>
    <element name="conversation" type="tns:moreMe"/>
  </sequence>
</complexType>
<complexType name="moreMe">
  <sequence>
    <element name="item" type="tns:allAboutMe"
             maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Example 19 shows the how Artix maps the recursive type into the CORBA type map of an Artix contract.

Example 19: *Recursive CORBA Typemap*

```
<corba:anonsequence name="moreMe" bound="0"
                    elemtype="ns1:allAboutMe"
   type="xsd1:moreMe"/>
<corba:struct name="allAboutMe"
              repositoryID="IDL:allAboutMe:1.0"
              type="xsd1:allAboutMe">
  <corba:member name="shoeSize" idltype="corba:long"/>
  <corba:member name="mated" idltype="corba:boolean"/>
  <corba:member name="conversation"
   idltype="ns1:moreMe"/>
</corba:struct>
```

While the XML in the CORBA typemap does not explicitly retain the recursive nature of recursive XMLSchema types, the IDL generated from the typemap restores the recursion in the IDL type. The IDL generated from the type map in Example 19 defines `allAboutMe` using recursion. Example 20 shows the generated IDL.

Example 20: *IDL for a Recursive Data Type*

```
\\IDL
struct allAboutMe
{
  long shoeSize;
  boolean mated;
  sequence<allAboutMe> conversation;
};
```

corba:anonstring

Synopsis

```
<corba:anonstring name="..." bound="..." type="..." />
```

Description

The `corba:anonstring` element is used to represent instances of anonymous XMLSchema simple types that are derived from `xsd:string`. As with `corba:anonsequence` elements, `corba:anonstring` elements do not result in generated IDL types.

Attributes

`corba:anonstring` elements have three attributes.

`name`	A unique identifier used to reference the CORBA type in the binding.
`bound`	The maximum length of the string.
`type`	The XMLSchema type of the base type. Typically this is `xsd:string`.

Examples

The complex type, `madAttr`, described in Example 21 contains a member, `style`, that is an instance of an anonymous type derived from `xsd:string`.

Example 21: *madAttr XML Schema*

```
<complexType name="madAttr">
  <sequence>
    <element name="style">
      <simpleType>
        <restriction base="xsd:string">
          <maxLength value="3"/>
        </restriction>
      </simpleType>
    </element>
    <element name="gender" type="xsd:byte"/>
  </sequence>
</complexType>
```

`madAttr` would generate the CORBA typemap shown in Example 22. Notice that `style` is given an IDL type defined by a `corba:anonstring` element.

Example 22: *madAttr CORBA typemap*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/anonCat/corba/typemap/">
  <corba:struct name="madAttr" repositoryID="IDL:madAttr:1.0" type="xsd1:madAttr">
    <corba:member idltype="ns1:styleType" name="style"/>
    <corba:member idltype="corba:char" name="gender"/>
  </corba:struct>
  <corba:anonstring bound="3" name="styleType" type="xsd:string"/>
</corba:typeMapping>
```

corba:object

Synopsis

```
<corba:object binding="..." name="..." repositoryID="..."
type="..." />
```

Description

The `corba:object` element is used to represent Artix references in the CORBA type map.

Arguments

`corba:object` elements have four attributes:

`binding`	Specifies the binding to which the object refers. If the annotation element is left off the reference declaration in the schema, this attribute will be blank.
`name`	Specifies the name of the CORBA type. If the annotation element is left off the reference declaration in the schema, this attribute will be `Object`. If the annotation is used and the binding can be found, this attribute will be set to the name of the interface that the binding represents.
`repositoryID`	Specifies the repository ID of the generated IDL type. If the annotation element is left off the reference declaration in the schema, this attribute will be set to `IDL:omg.org/CORBA/Object/1.0`. If the annotation is used and the binding can be found, this attribute will be set to a properly formed repository ID based on the interface name.
`type`	Specifies the schema type from which the CORBA type is generated. This attribute is always set to `references:Reference`.

Examples

Example 23 shows an Artix contract fragment that uses Artix references.

Example 23: *Reference Sample*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="bankService"
 targetNamespace="http://schemas.myBank.com/bankTypes"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.myBank.com/bankService"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://schemas.myBank.com/bankTypes"
 xmlns:corba="http://schemas.iona.com/bindings/corba"
 xmlns:corbatm="http://schemas.iona.com/typemap/corba/bank.idl"
 xmlns:references="http://schemas.iona.com/references">
  <types>
    <schema
     targetNamespace="http://schemas.myBank.com/bankTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:import schemaLocation="./references.xsd"

   namespace="http://schemas.iona.com/references"/>
```

Example 23: *Reference Sample (Continued)*

```
...
      <xsd:element name="account" type="references:Reference">
        <xsd:annotation>
          <xsd:appinfo>
          corba:binding=AccountCORBABinding
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:element>
  </schema>
</types>
...
  <message name="find_accountResponse">
    <part name="return" element="xsd1:account"/>
  </message>
  <message name="create_accountResponse">
    <part name="return" element="xsd1:account"/>
  </message>
  <portType name="Account">
    <operation name="account_id">
      <input message="tns:account_id" name="account_id"/>
      <output message="tns:account_idResponse"
              name="account_idResponse"/>
    </operation>
    <operation name="balance">
      <input message="tns:balance" name="balance"/>
      <output message="tns:balanceResponse"
              name="balanceResponse"/>
    </operation>
    <operation name="withdraw">
      <input message="tns:withdraw" name="withdraw"/>
      <output message="tns:withdrawResponse"
              name="withdrawResponse"/>
      <fault message="tns:InsufficientFundsException"
   name="InsufficientFunds"/>
    </operation>
    <operation name="deposit">
      <input message="tns:deposit" name="deposit"/>
      <output message="tns:depositResponse"
              name="depositResponse"/>
    </operation>
  </portType>
  <portType name="Bank">
    <operation name="find_account">
      <input message="tns:find_account" name="find_account"/>
      <output message="tns:find_accountResponse"
              name="find_accountResponse"/>
      <fault message="tns:AccountNotFound"
             name="AccountNotFound"/>
    </operation>
```

Example 23: *Reference Sample (Continued)*

```
    <operation name="create_account">
      <input message="tns:create_account"
   name="create_account"/>
      <output message="tns:create_accountResponse"
              name="create_accountResponse"/>
      <fault message="tns:AccountAlreadyExistsException"
             name="AccountAlreadyExists"/>
    </operation>
  </portType>
</definitions>
```

The element named `account` is a reference to the interface defined by the `Account` port type and the `find_account` operation of `Bank` returns an element of type `account`. The annotation element in the definition of `account` specifies the binding, `AccountCORBABinding`, of the interface to which the reference refers.

Example 24 shows the generated CORBA typemap resulting from generating both the `Account` and the `Bank` interfaces into the same contract.

Example 24: *CORBA Typemap with References*

```
<corba:typeMapping
  targetNamespace="http://schemas.myBank.com/bankService/corba/typemap/">
...
  <corba:object binding="" name="Object"
                repositoryID="IDL:omg.org/CORBA/Object/1.0"
   type="references:Reference"/>
  <corba:object binding="AccountCORBABinding" name="Account"
                repositoryID="IDL:Account:1.0" type="references:Reference"/>
</corba:typeMapping>
```

There are two entries because `wsdltocorba` was run twice on the same file. The first CORBA object is generated from the first pass of `wsdltocorba` to generate the CORBA binding for `Account`. Because `wsdltocorba` could not find the binding specified in the annotation, it generated a generic `Object` reference. The second CORBA object, `Account`, is generated by the second pass when the binding for `Bank` was generated. On that pass, `wsldtocorba` could inspect the binding for the `Account` interface and generate a type-specific object reference.

Example 25 shows the IDL generated for the `Bank` interface.

Example 25: *IDL Generated From Artix References*

```
//IDL
...
interface Account
{
  string account_id();
  float balance();
  void withdraw(in float amount)
    raises(::InsufficientFundsException);
  void deposit(in float amount);
};
```

Example 25: *IDL Generated From Artix References (Continued)*

```
interface Bank
{
  ::Account find_account(in string account_id)
    raises(::AccountNotFoundException);
  ::Account create_account(in string account_id,
                           in float initial_balance)
    raises(::AccountAlreadyExistsException);
};
```

Tuxedo FML Binding

Artix supports the use of Tuxedo's FML buffers. It uses a set of Artix specific elements placed in the WSDL binding element.

Runtime Compatibility

The Tuxedo FML extension elements are only compatible with the C++ runtime.

Namespace

The WSDL extensions used for the FML binding are defined in the namespace `http://schemas.iona.com/transports/tuxedo`. Add the following namespace declaration to any contracts that use an FML binding:

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

FML\XMLSchema Support

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

Table 3: *FML Type Support*

XML Schema Type	FML Type
xsd:short	short
xsd:unsignedShort	short
xsd:int	long
xsd:unsignedInt	long
xsd:float	float
xsd:double	double
xsd:string	string
xsd:base64Binary	string
xsd:hexBinary	string

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

tuxedo:binding

Synopsis

```
<tuxedo:binding />
```

Description

The `tuxedo:binding` element informs Artix that the payload being described is an FML buffer. It is a child of the WSDL `binding` element and has no children.

tuxedo:fieldTable

Synopsis

```
<tuxedo:fieldTable type="...">
  <tuxedo:field ... />
  ...
</tuxedo:fieldTable>
```

Description

The `tuxedo:fieldTable` element contains the mappings between the elements defined in the logical section of the contract and their associated FML `fieldid`.

Attributes

The `tuxedo:fieldTable` element has one required attribute, `type`, that specifies if the FML buffer is an FML16 buffer or an FML32 buffer. Table 4 shows the values of the type attribute.

Table 4: *Values of tuxedo:fieldTable Element's type Attribute*

Value	Meaning
`FML`	The represented FML buffer is a FML16 buffer.
`FML32`	The represented FML buffer is an FML32 buffer.

tuxedo:field

Synopsis

```
<tuxedo:field name="..." id="..." />
```

Description

The `tuxedo:field` element defines the association between an element in the logical contract and its corresponding entry in the physical FML buffer. Each element in a message, either a message part or an element in a complex type, must have a corresponding `tuxedo:field` element in the FML binding.

Attributes

The `tuxedo:field` element takes two attributes:

`name` — The value of the `name` attribute from the logical message element to which this `tuxedo:field` element corresponds.

`id` — The `fieldId` value of the corresponding element in the generated C++ header defining the FML buffer.

tuxedo:operation

Synopsis

```
<tuxedo:operaiton />
```

Description

The `tuxedo:operation` element is a child of the WSDL binding's `operation` element. It informs Artix that the messages used by the operation are being passed as FML buffers.

Fixed Binding

The fixed binding supports mapping between XML Schema message definitions and messages formatted in fixed length records.

Runtime Compatibility

The fixed binding's extension elements are only compatible with the C++ runtime.

Namespace

The extensions used to describe fixed record length messages 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

Synopsis

```
<fixed:binding justification="..." encoding="..."
               padHexCode="..." />
```

Description

The `fixed:binding` element is a child of the WSDL `binding` element. It specifies that the binding defines a mapping between fixed record length data and the XMLSchema representation of the data.

Attributes

The `fixed:binding` element has three attributes:

`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 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 the `fixed:binding` element become the default settings for all the messages being mapped to the current binding.

fixed:operation

Synopsis

```
<fixed:operation discriminator="..." />
```

Description

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

Attributes

The `fixed:operation` element 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 operation 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.

fixed:body

Synopsis

```
<fixed:body justification="..." encoding="..." padHexCode="...">
  ...
</fixed:body>
```

Description

The `fixed:body` element is a child element of the `input`, `output`, and `fault` 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.

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 WSDL `message` element defining the logical message.

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.
- `fixed:sequence` maps message parts defined using a `sequence` complex type.

Note: Complex types defined using `all` are not supported by the fixed binding.

- `fixed:choice` maps message parts defined using a `choice` complex type.

Attributes

The `fixed:body` element has three attributes:

`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.

fixed:field

Synopsis

```
<fixed:field name="..." "size="..." format="..."
             justification="..." fixedValue="..." bindingOnly="...">
  <fixed:enumeration ... />
  ...
</fixed:field>
```

Description

The `fixed:field` element is used to map simple data types to a field in a fixed record length message. It is the child of a `fixed:body` element.

Attributes

The `fixed:field` element has the following attributes:

`name`	Specifies the name of the logical message part that this element represents. It is a required attribute.
`size`	Specifies the maximum number of characters in a message part whose base type is `xsd:string`. Also used to specify the number of characters in the on-wire values used to represent the values of an enumerated type. For more information see "fixed:enumeration" on page 43.
`format`	Specifies how non-string data is formatted when it is placed on the wire. For numerical data, formats are entered using `#` to represent numerical fields and `.` to represent decimal places. For example `##.##` would be used to represent `12.04`. Also can be used for string data that is a date. Date formats use the standard date format syntax. For example, `mm/dd/yy` would represent dates such as `02/23/04` and `11/02/98`.
`justification`	Specifies the default justification of the data contained in the field. Valid values are `left` and `right`. Default is `left`.
`fixedValue`	Specifies the value to use for the represented logical message part. The value of `fixedValue` is always the value placed on the wire for the represented message part. It will override any values set in the application code.
`bindingOnly`	Specifies if the field appears in the logical definition of the message. The default value is `false`. When set to `true`, this attribute signals Artix that it needs to insert a field into the on-wire message that does not appear in the logical message. `bindingOnly` is used in conjunction with the `fixedValue` attribute. The `fixedValue` attribute is used to specify the data to be written into the binding-only field.

Examples

The following examples show different ways of representing data using a `fixed:field` element:

- String data
- Numeric data
- Dates
- Binding only records

String data

The logical message part, `raverID`, described in Example 26 would be mapped to a `fixed:field` similar to Example 27.

Example 26: *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 27: *Fixed String Mapping*

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

Numeric data

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 28.

Example 28: *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 you would specify its `format` with `##.#`, as shown in Example 29. This provides Artix with the metadata needed to properly handle the data.

Example 29: *Mapping Numerical Data to a Fixed Binding*

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

Dates

Dates are specified in a similar fashion. For example, the `format` of 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 30.

Example 30: *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 31.

Example 31: *Fixed Format Date Mapping*

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

Binding only records

If you were sending reports that included a fixed expiration date that you did not want exposed to the application, you could create a binding only record called expDate. It would be mapped to the fixed field shown in Example 32.

Example 32: *fixedValue Mapping*

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

fixed:enumeration

Synopsis

```
<fixed:enumeration value="..." fixedValue="..." />
```

Description

The fixed:enumeration element is a child of a fixed:body element. It is used to represent the possible values of an enumerated type and define how those values are represented on the wire.

Attributes

The fixed:enumeration element has two required attributes:

value	Is the value of the corresponding enumeration value in the logical description of the message part.
fixedValue	Specifies the string value that will be used to represent the logical value on the wire. The length of the string used is determined by the value of the parent fixed:field element's length attribute.

Examples

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 33.

Example 33: *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>
```

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 definitions or some other value. The enumerated type in Example 33 could be

mapped to the fixed field shown in Example 34. Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to `OrangeTango`.

Example 34: *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>
```

fixed:choice

Synopsis

```
<fixed:choice name="..." discriminatorName="...">
  <fixed:case ... >
    ...
  </fixed:case>
  ...
</fixed:choice>
```

Description

The `fixed:choice` element is a child of a fixed:body element. It maps choice complex types to a field in a fixed record length message. The actual values of the choice are defined using fixed:case child elements. A `fixed:choice` element must have a `fixed:case` child element for each possible value defined in the choice complex type it represents.

Attributes

The `fixed:choice` element has the following attributes:

`name`	Specifies the name of the logical message part the choice element is mapping. This attribute is required.
`discriminatorName`	Specifies the name of a binding-only field that is used as the discriminator for the union. The binding-only field must defined as part of the parent `fixed:body` element and must be capable of representing the discriminator.

fixed:case

Synopsis

```
<fixed:case name="..." fixedValue="...">
  ...
</fixed:case>
```

Description

The `fixed:case` element is a child of the fixed:choice element. It describes the complete mapping for an element of a choice complex type to a field in a fixed record length message.

To fully describe how the logical data that is represented by a `fixed:case` element is mapped into a field in a fixed record length message, you need to create a mapping for the logical element using children to the `fixed:case` element. 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. fixed:field

elements describe simple types. fixed:choice elements describe choice complex types. fixed:sequence elements describe sequence complex types.

Attributes

The `fixed:case` element has the following required attributes:

`name`	Specifies the value of the `name` attribute of the corresponding element in the choice complex type being mapped.
`fixedValue`	Specifies the discriminator value that selects this case. If the parent `fixed:choice` element has its `discriminatorName` attribute set, the value must conform to the format specified for that field.

Examples

Example 35 shows an Artix contract fragment mapping a choice complex type to a fixed record length message.

Example 35: *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"/>
```

Example 35: *Mapping a Union to a Fixed Record Length Message (Continued)*

```
    <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

Synopsis

```
<fixed:sequence name="..." occurs="..." counterName="...">
  ...
</fixed:field>
```

Description

The `fixed:sequence` element can be a child to a fixed:body element, a `fixed:case` element, or another `fixed:sequence` element. It maps a sequence complex type to a field in a fixed record length message.

To fully describe how the complex type that is represented by a `fixed:sequence` element is mapped into a field in a fixed record length message, you need to create a mapping for each of the complex type's elements using children to the `fixed:sequence` element. 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. fixed:field elements describe simple types. fixed:choice elements describe choice complex types. fixed:sequence elements describe sequence complex types.

Attributes

The `fixed:sequence` element has the following attributes:

`name`	Specifies the value of the `name` attribute from the corresponding logical complex type. This attribute is required.
`occurs`	Specifies the number of times this sequence occurs in the message buffer. This value corresponds the value of the `maxOccurs` attribute of the corresponding logical complex type.
`counterName`	Specifies the name of the binding-only field that is used to store the actual number of times this sequence occurs in the on-wire message. The corresponding fixed:field element must have enough digits to hold the any whole number up the value of the `occurs` attribute.

Examples

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 36 shows an Artix contract fragment for such a mapping.

Example 36: *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:sequence>
   </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>
```

Tagged Binding

The tagged binding maps between XMLSchema message definitions and self-describing, variable record length messages.

Runtime Compatibility

The tagged binding's extension elements are only compatible with the C++ runtime.

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

Synopsis

```
<tagged:binding selfDescribing="..." fieldSeperator="..."
                fieldNameValueSeperator="..." scopeType="..."
                flattened="..." messageStart="..." messageEnd="..."
                unscopedArrayElement="..." ignoreUnknownElement="..."
                ignoreCase="..." />
```

Description

The `tagged:binding` element specifies that the binding maps logical messages to tagged data messages.

Attributes

The `tagged:binding` element has the following ten attributes:

`selfDescribing`	Specifies 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. This attribute is required.
`fieldSeparator`	Specifies the delimiter the message uses to separate fields. Valid values include any character that is not a letter or a number. This attribute is required.
`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 that require a special character at the start of a the data sequence. Valid values include any character that is not a letter or a number.
`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 element 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

Synopsis

```
<tagged:operation discriminator="..." discrininatorStyle="..." />
```

Description

The `tagged:operation` element is a child element of the WSDL `operation` element. It specifies that the operation's messages are being mapped to a tagged data message.

Attributes

The `tagged:operation` element takes two optional attributes:

`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

Synopsis

```
<tagged:body>
  ...
</tagged:body>
```

Description

The `tagged:body` element 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.

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

- tagged:field
- tagged:sequence
- tagged:choice

The children describe the detailed mapping of the XMLSchema message to the tagged data to be sent on the wire.

tagged:field

Synopsis

```
<tagged:field name="..." alias="...">
  <tagged:enumeration ... />
  ...
</tagged:field>
```

The `tagged:field` element is a child of a tagged:body element. It maps simple types and enumerations to a field in a tagged data message. When describing enumerated types a `tagged:field` element will have one or more tagged:enumeration child elements.

Attributes

The `tagged:field` element has two attributes:

`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.

tagged:enumeration

Synopsis

```
<tagged:enumeration value="..." />
```

Description

The `tagged:enumeration` element is a child element of a tagged:field element. It is used to map the value of an enumerated types to a field in a tagged data message.

Parameters

The `tagged:enumeration` element has one required attribute, `value`, that corresponds to the enumeration value as specified in the logical description of the enumerated type.

Examples

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 37.

Example 37: *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 a tagged data field as shown in Example 38.

Example 38: *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

Synopsis

```
<tagged:sequence name="..." alias="..." occurs="...">
  ...
</tagged:sequence>
```

Description

The `taggeded:sequence` element is a child of a tagged:body element, a `tagged:sequence` element, or a tagged:case element. It maps arrays and sequence complex types to fields in a tagged data message. A `tagged:sequence` element contains one or more children to map the corresponding logical type's parts to fields in a tagged data message. The child elements can be of the following types:

- tagged:field
- tagged:sequence
- tagged:choice

Attributes

The `taggeded:sequence` element has three attributes:

`name`	Specifies the name of the logical message part that is being mapped into the tagged data message. This is a required attribute.
`alias`	Specifies an alias for the sequence that can be used to identify it on the wire.
`occurs`	Specifying the number of times the sequence appears. This attribute is used to map arrays.

Examples

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 39 shows an Artix

contract fragment for such a mapping.

Example 39: *Mapping a Sequence to a Tagged Data Format*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="taggedDataMappingsample"
   targetNamespace="http://www.iona.com/taggedService"
    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="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: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

Synopsis

```
<tagged:choice name="..." discriminatorName="..." alais="...">
  <tagged:case ...>
  ...
</tagged:choice>
```

The `tagged:choice` element is a child of a tagged:body element, a tagged:sequence element, or a tagged:case element. It maps unions to a field in a tagged data message. A `tagged:choice` element may contain one or more tagged:case child elements to map the cases for the union to a field in a tagged data message.

Parameters

The `tagged:choice` element has three attributes:

`name`	Specifies the name of the logical message part being mapped into the tagged data message. This is a required attribute.
`discriminatorName`	Specifies the message part used as the discriminator for the union.
`alias`	Specifies an alias for the union that can be used to identify it on the wire.

tagged:case

Synopsis

```
<tagged:case value="..." />
```

Description

The `tagged:case` element is a child element of a `tagged:choice` element. It describes the complete mapping of a union's individual cases to a field in a tagged data message. A `tagged:case` element must have one child element to describe the mapping of the case's data to a field, or fields, to a tagged data message. Valid child elements are tagged:field, tagged:sequence, and tagged:choice.

Attributes

The `tagged:case` element has one required attribute, `name`, that corresponds to the name of the case element in the union's logical description.

Examples

Example 40 shows an Artix contract fragment mapping a union to a tagged data format.

Example 40: *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/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="tagUnion">
  <part name="stationPart" type="tns:unionStationType"/>
</message>
<portType name="tagUnionPortType">
...
</portType>
```

Example 40: *Mapping a Union to a Tagged Data Format*

```
<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>
```

XML Binding

Artix includes a binding that supports the exchange of XML documents without the overhead of a SOAP envelope.

Runtime Compatibility

The XML binding's extensions are compatible with the C++ runtime.

Namespace

The extensions used to describe XML format bindings are defined in the namespace `http://celtix.objectweb.org/bindings/xmlformat`. Artix 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"
```

xformat:binding

Synopsis

```
<xformat:binding rootNode="..." />
```

Description

The `xformat:binding` element is the child of the WSDL `binding` element. It signifies that the messages passing through this binding will be sent as XML documents without a SOAP envelope.

Attributes

The `xformat:binding` element has a single optional attribute called `rootNode`. The `rootNode` attribute specifies the QName for the element that serves as the root node for the XML document generated by Artix. When the `rootNode` attribute is not set, Artix 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 RCP style messages.

xformat:body

Synopsis

```
<xformat:body rootNode="..." />
```

Description

The `xformat:body` element is an optional child of the WSDL `input` element, the WSDL `output` element, and the WSDL `fault` element. It is used to override the value of the `rootNode` attribute specified in the binding's xformat:binding element.

Attributes

The `xformat:body` element has a single attribute called `rootNode`. The `rootNode` attribute specifies the QName for the element that serves as the root node for the XML document generated by Artix. When the `rootNode` attribute is not set, Artix 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 RCP style messages.

Pass Through Binding

The pass through binding is a simple binding that passes blobs through the message layers. The application level code must know how to handle the incoming data.

Runtime Compatibility

The pass through binding's extension elements are only compatible with the C++ runtime.

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

tagged:binding

Synopsis

```
<passthru:binding />
```

Description

The `passthru:binding` element specifies that the binding passes the message through as a blob.

Part II

Ports

In this part

This part contains the following chapters:

HTTP Port

Along with the standard WSDL elements used to specify the location of an HTTP port, Artix uses a number of extensions for fine tuning the configuration of an HTTP port.

Standard WSDL Elements

http:address

Synopsis

```
<http:address location="..." />
```

Description

The `http:address` element is a child of the WSDL `port` element. It specifies the address of the HTTP port of a service that is not using SOAP messages to communicate.

Attributes

The `http:address` element has a single required attribute called `location`. The `location` attribute specifies the service's address as a URL.

soap:address

Synopsis

```
<soap:address location="..." />
```

Description

The `soap:address` element is a child of the WSDL `port` element. It specifies the address of the HTTP port of a service that uses SOAP 1.1 messages to communicate.

Attributes

The `soap:address` element has a single required attribute called `location`. The `location` attribute specifies the service's address as a URL.

wsoap12:address

Synopsis

```
<wsoap12:address location="..." />
```

Description

The `wsoap12:address` element is a child of the WSDL `port` element. It specifies the address of the HTTP port of a service that uses SOAP 1.2 messages to communicate.

Attributes

The `wsoap12:address` element has a single required attribute called `location`. The `location` attribute specifies the service's address as a URL.

Configuration Extensions for C++

Namespace

Example 41 shows the namespace entries you need to add to the `definitions` element of your contract to use the Artix C++ runtime's HTTP extensions.

Example 41: *Artix HTTP Extension Namespaces*

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

http-conf:client

Synopsis

```
<http-conf:client SendTimeout="..." RecieveTimeout="..."
                  AutoRedirect="..." UserName="..."
                  Password="..." AuthorizationType="..."
                  Authorization="..." Accept="..."
                  AcceptLanguage="..." AcceptEncoding="..."
                  ContentType="..." Connection="..."
                  Host="..." ConnectionAttempts="..."
                  CacheControl="..." Cookie="..."
                  BrowserType="..." Refferer="..."
                  ProxyServer="..." ProxyUsername="..."
                  ProxyPassword="..." ProxyAuthorizationType="..."
                  ProxyAuthorization="..." UseSecureSockets="..."
                  ClientCertificates="..." ClientCertificateChain="..."
                  ClientPrivateKey="..." ClientPrivateKeyPassword="..."
                  TrustedRootCertificate="..." />
```

Description

The `http-conf:client` element is a child of the WSDL `port` element. It is used to specify client-side configuration details.

Attributes

The `http-conf:client` element has the following attributes:

`SendTimeout`	Specifies the length of time, in milliseconds, the client tries to send a request to the server before the connection is timed out. Default is `30000`.
`ReceiveTimeout`	Specifies the length of time, in milliseconds, the client tries to receive a response from the server before the connection is timed out. The default is `30000`.
`AutoRedirect`	Specifies if a request should be automatically redirected when the server issues a redirection reply via `RedirectURL`. The default is `false`, to let the client redirect the request itself.

UserName	Specifies the user name that the client will use for authentication with a service. This value is passed as an attribute in each request's transport header.
Password	Specifies the password that the client will use for authentication with a service. This value is passed as an attribute in each request's transport header.
AuthorizationType	Specifies the name of the authorization scheme the client wishes to use.
Authorization	Specifies the authorization credentials used to perform the authorization.
Accept	Specifies what media types the client is prepared to handle.
AcceptLanguage	Specifies the client's preferred language for receiving responses.
AcceptEncoding	Specifies what content codings the client is prepared to handle.
ContentType	Specifies the media type of the data being sent in the body of the client request.
Host	Specifies the Internet host and port number of the resource on which the client request is being invoked.
Connection	Specifies if the client wants a particular connection to be kept open after each request/response dialog.
ConnectionAttempts	Specifies the number of times a client will transparently attempt to connect to server.
CacheControl	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.
Cookie	Specifies a static cookie to be sent to the server along with all requests.
BrowserType	Specifies information about the browser from which the client request originates.
Referer	Specifies the URL of the resource that directed the client to make requests on a particular service.
ProxyServer	Specifies the URL of the proxy server, if one exists along the message path.
ProxyUserName	Specifies the username to use for authentication on the proxy server if it requires separate authorization.

ProxyPassword	Specifies the password to use for authentication on the proxy server if it requires separate authorization.
ProxyAuthorizationType	Specifies the name of the authorization scheme used with the proxy server.
ProxyAuthorization	Specifies the authorization credentials used to perform the authorization with the proxy server.
UseSecureSockets	Indicates if the client wants to open a secure connection.
ClientCertificate	Specifies the full path to the PKCS12-encoded X509 certificate issued by the certificate authority for the client.
ClientCertificateChain	Specifies the full path to the file that contains all the certificates in the chain.
ClientPrivateKey	Specifies the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by ClientCertificate.
ClientPrivateKeyPassword	Specifies a password that is used to decrypt the PKCS12-encoded private key.
TrustedRootCertificate	Specifies the full path to the PKCS12-encoded X509 certificate for the certificate authority.

http-conf:server

Synopsis

```
<http_conf:server SendTimeout="..." RecieveTimeout="..."
                  SurpressClientSendErrors="..."
                  SurpressClientRecieveErrors="..."
                  HonnorKeepAlive="..." RedirectURL="..."
                  CacheControl="..." ContentLocation="..."
                  ContentType="..." ContentEncoding="..."
                  ServerType="..." UseSecureSockets="..."
                  ServerCertificate="..." ServerCertificateChain="..."
                  ServerPrivateKey="..." ServerPrivateKeyPassword="..."
                  TrustedRootCertificate="..." />
```

Description

The http-conf:server element is a child of the WSDL port element. It is used to specify server-side configuration details.

Attributes

The http-conf:server element has the following attributes:

SendTimeout	Sets the length of time, in milliseconds, the server tries to send a response to the client before the connection times out. The default is 30000.

ReceiveTimeout	Sets the length of time, in milliseconds, the server tries to receive a client request before the connection times out. The default is 30000.
SuppressClientSendErrors	Specifies whether exceptions are to be thrown when an error is encountered on receiving a client request. The default is false; exceptions are thrown on encountering errors.
SuppressClientReceiveErrors	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a client. The default is false; exceptions are thrown on encountering errors.
HonorKeepAlive	Specifies whether the server honors client requests for a connection to remain open after a response has been sent. The default is Keep-Alive; Keep-alive requests are honored. false specifies that keep-alive requests are ignored.
RedirectURL	Sets 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.
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a server to a client.
ContentLocation	Sets the URL where the resource being sent in a server response is located.
ContentType	Sets the media type of the information being sent in a server response, for example, text/html or image/gif.
ContentEncoding	Specifies what additional content codings have been applied to the information being sent by the server.
ServerType	Specifies what type of server is sending the response to the client. Values take the form *program-name/version*. For example, Apache/1.2.5.
UseSecureSockets	Indicates whether the server wants a secure HTTP connection running over SSL or TLS.

ServerCertificate	Sets the full path to the PKCS12-encoded X509 certificate issued by the certificate authority for the server.
ServerCertificateChain	Sets the full path to the file that contains all the certificates in the server's certificate chain.
ServerPrivateKey	Sets the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by ServerCertificate.
ServerPrivateKeyPassword	Sets a password that is used to decrypt the PKCS12-encoded private key, if it has been encrypted with a password.
TrustedRootCertificate	Sets the full path to the PKCS12-encoded X509 certificate for the certificate authority. This is used to validate the certificate presented by the client.

Attribute Details

AuthorizationType

Description

The AuthorizationType attribute corresponds to the HTTP AuthorizationType property. It specifies the name of the authorization scheme the client wishes to use. This information is specified and handled at the application level. Artix does not perform any validation on this value. It is the user's responsibility to ensure that the correct scheme name is specified, as appropriate.

Note: If the client wants to use basic username and password-based authentication this does not need to be set.

Authorization

Description

The Authorization attribute corresponds to the HTTP Authorization property. It specifies the authorization credentials the client wants the server to use when performing the authorization. The credentials are encoded and handled at the application-level. Artix does not perform any validation on the specified value. It is the user's responsibility to ensure that the correct authorization credentials are specified, as appropriate.

Note: If the client wants to use basic username and password-based authentication this does not need to be set.

Accept

Description

The Accept attribute corresponds to the HTTP Accept property. It specifies what media types the client is prepared to handle. The

value of the attribute is specified using as multipurpose internet mail extensions (MIME) types.

MIME type values

MIME types are regulated by the Internet Assigned Numbers Authority (IANA). They consist of a main type and sub-type, separated by a forward slash. For example, a main type of `text` might be qualified as follows: `text/html` or `text/xml`. Similarly, a main type of image might be qualified as follows: `image/gif` or `image/jpeg`.

An asterisk (*) can be used as a wildcard to specify a group of related types. For example, if you specify `image/*`, this means that the client can accept any image, regardless of whether it is a GIF or a JPEG, and so on. A value of `*/*` indicates that the client is prepared to handle any type.

Examples of typical types that might be set are:

- `text/xml`
- `text/html`
- `text/text`
- `image/gif`
- `image/jpeg`
- `application/jpeg`
- `application/msword`
- `application/xbitmap`
- `audio/au`
- `audio/wav`
- `video/avi`
- `video/mpeg`

See Also

See http://www.iana.org/assignments/media-types/ for more details.

AcceptLanguage

Description

The `AcceptLanguage` attribute corresponds to the HTTP AcceptLanguage property. It specifies what language (for example, American English) the client prefers for the purposes of receiving a response.

Specifying the language

Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, `en-US` represents American English.

See Also

A full list of language codes is available at http://www.w3.org/WAI/ER/IG/ert/iso639.htm.

A full list of country codes is available at http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html.

AcceptEncoding

Description

The `AcceptEncoding` attribute corresponds to the HTTP AcceptEncoding Property. It specifies what content encodings the client is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible

content encoding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`.

The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Artix performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.

See Also

See http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html for more details on content encodings.

ContentType

Description

The `ContentType` attribute corresponds to the HTTP ContentType property. It specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types.

MIME type values

MIME types are regulated by the Internet Assigned Numbers Authority (IANA). MIME types consist of a main type and sub-type, separated by a forward slash. For example, a main type of `text` might be qualified as follows: `text/html` or `text/xml`. Similarly, a main type of image might be qualified as follows: `image/gif` or `image/jpeg`.

The default type is `text/xml`. Other specifically supported types include:

- `application/jpeg`
- `application/msword`
- `application/xbitmap`
- `audio/au`
- `audio/wav`
- `text/html`
- `text/text`
- `image/gif`
- `image/jpeg`
- `video/avi`
- `video/mpeg`.

Any content that does not fit into any type in the preceding list should be specified as `application/octet-stream`.

Client settings

For clients this attribute is only relevant if the client request specifies the `POST` method to send data to the server for processing.

For web services, this should be set to `text/xml`. If the client is sending HTML form data to a CGI script, this should be set to `application/x-www-form-urlencoded`. If the HTTP `POST` request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to `application/octet-stream`.

See Also

See http://www.iana.org/assignments/media-types/ for more details.

ContentEncoding

Description

The `ContentEncoding` attribute corresponds to the HTTP ContentEncoding property. This property specifies any additional content encodings that have been applied to the information being sent by the server. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`.

The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Artix performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.

See Also

See http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html for more details on content encodings.

Host

Description

The `Host` attribute corresponds to the HTTP Host property. It specifies the internet host and port number of the resource on which the client request is being invoked. This attribute is typically not required. Typically, this attribute does not need to be set. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same internet protocol (IP) address).

Connection

Description

The `Connection` attribute specifies whether a particular connection is to be kept open or closed after each request/response dialog. Valid values are `close` and `Keep-Alive`. The default, `Keep-Alive`, specifies that the client want to keep its connection open after the initial request/response sequence. If the server honors it, the connection is kept open until the client closes it. `close` specifies that the connection to the server is closed after each request/response sequence.

CacheControl

Description

The `CacheControl` attribute specifies directives about the behavior of caches involved in the message chain between clients and servers. The attribute is used for both client and server. However, clients and servers have different settings for specifying cache behavior.

Client-side

Table 5 shows the valid settings for `CacheControl` in `http-conf:client`.

Table 5: *Settings for http-conf:client CacheControl*

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.

Server-side

Table 6 shows the valid values for `CacheControl` in `http-conf:server`.

Table 6: *Settings for http-conf:server CacheControl*

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 revaildate expired entries that relate to a response before that entry can be used in a subsequent response.
`proxy-revelidate`	Means the same as `must-revalidate`, except that it can only be enforced on shared caches and is ignored by private unshared caches. If using this directive, the `public` cache directive must also be used.
`max-age`	Clients can accept a response whose age is no greater that the specified number of seconds.
`s-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.

Table 6: *Settings for http-conf:server CacheControl*

Directive	Behavior
`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.

BrowserType

Description

The `BrowserType` attribute specifies information about the browser from which the client request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based upon the client that is sending the request.

Referer

The `Referer` attribute corresponds to the HTTP Referer property. It specifies the URL of the resource that directed the client to make requests on a particular service. Typically this HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications.

If the `AutoRedirect` attribute is set to `true` and the client request is redirected, any value specified in the `Referer` attribute is overridden. The value of the HTTP Referer property will be set to the URL of the service who redirected the client's original request.

ProxyServer

Description

The `ProxyServer` attribute specifies the URL of the proxy server, if one exists along the message path. A proxy can receive client requests, possibly modify the request in some way, and then forward the request along the chain possibly to the target server. A proxy can act as a special kind of security firewall.

Note: Artix does not support the existence of more than one proxy server along the message path.

ProxyAuthorizationType

Description

The `ProxyAuthorizationType` attribute specifies the name of the authorization scheme the client wants to use with the proxy server. This name is specified and handled at application level. Artix does not perform any validation on this value. It is the user's responsibility to ensure that the correct scheme name is specified, as appropriate.

Note: If basic username and password-based authentication is being used by the proxy server, this does not need to be set.

ProxyAuthorization

Description

The `ProxyAuthorization` attribute specifies the authorization credentials the client will use to perform authorization with the proxy server. These are encoded and handled at application-level. Artix does not perform any validation on the specified value. It is the user's responsibility to ensure that the correct authorization credentials are specified, as appropriate.

Note: If basic username and password-based authentication is being used by the proxy server, this does not need to be set.

UseSecureSockets

Description

The `UseSecureSockets` attribute indicates if the application wants to open a secure connection using SSL or TLS. A secure HTTP connection is commonly referred to as HTTPS. Valid values are `true` and `false`. The default is `false`; the endpoint does not want to open a secure connection.

Note: If the `http:address` element's `location` attribute, or the `soap:address` element's `location` attribute, has a value with a prefix of `https://`, a secure HTTP connection is automatically enabled, even if `UseSecureSockets` is not set to `true`.

RedirectURL

Description

The `RedirectURL` attribute corresponds to the HTTP RedirectURL property. It specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to `302` and the status description is set to `Object Moved`.

ServerCertificateChain

Description

PKCS12-encoded X509 certificates can be issued by intermediate certificate authorities that are not trusted by the client, but which have had their certificates issued in turn by a trusted certificate authority. If this is the case, you can use the `ServerCertificateChain` attribute to allow the certificate chain of PKCS12-encoded X509 certificates to be presented to the client for verification. It specifies the full path to the file that contains all the certificates in the chain.

CORBA Port

Artix supports a robust mechanism for configuring a CORBA endpoint.

Runtime Compatibility

The CORBA transport's extension elements are compatible with the C++ runtime.

C++ Runtime Namespace

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

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

corba:address

Synopsis

```
<corba:address location="..."/>
```

Description

The `corba:address` element is a child of a WSDL `port` element. It specifies the IOR for the service's CORBA object.

Attributes

The `corba:address` element has one required attribute named `location`. The `location` attribute contains a string specifying the IOR. You have four options for specifying IORs in Artix contracts:

- Entering the object's IOR directly into the contract using the stringified IOR format:

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

- Entering a file location for the IOR using the following syntax:

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

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

- Entering the object's name using the `corbaname` format:

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

  When you use the `corbaname` format for specifying the IOR, Artix will look-up the object's IOR in the CORBA name service.

- Entering the port at which the service exposes itself, using the `corbaloc` syntax.

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

corba:policy

Synopsis

```
<corba:policy
poaname="..."|persistent="..."|serviceid="..." />
```

Description

The `corba:policy` element is a child of a WSDL `port` element. It specifies the POA polices the Artix service will use when creating the POA for connecting to a CORBA object. Each `corba:policy` element can only specify one policy. Therefore to define multiple policies you must use multiple `corba:policy` elements.

Attributes

The `corba:policy` element uses attributes to specify the policy it is describing. The following attributes are used:

`poaname`	Specifies the POA name to use when connecting to the CORBA object. The default POA name is `WS_ORB`.
`persistent`	Specifies the value of the POA's persistence policy. The default is `false`; the POA is not persistent.
`serviceid`	Specifies the value of the POA's ID. By default, Artix POAs are assigned their IDs by the ORB.

See Also

For more information about CORBA POA policies see the Orbix documentation.

IIOP Tunnel Port

The IIOP tunnel transport allows you to send non-CORBA data over IIOP. This allows you to use a number of the CORBA services.

Runtime Compatibility

The IIOP tunnel transport's extensions are only compatible with the C++ runtime.

Namespace

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

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

iiop:address

Synopsis

```
<iiop:address location="..."/>
```

Description

The `iiop:address` element is a child of a WSDL `port` element. It specifies the IOR for the CORBA object created for the service.

Attributes

The `iiop:address` element has one required attribute named `location`. The `location` attribute contains a string specifying the IOR. You have four options for specifying IORs in Artix contracts:

- Entering the object's IOR directly into the contract using the stringified IOR format:

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

- Entering a file location for the IOR using the following syntax:

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

  Note: The file specification needs three backslashes (///).

- Entering the object's name using the `corbaname` format:

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

  When you use the `corbaname` format for specifying the IOR, Artix will look-up the object's IOR in the CORBA name service.

- Entering the port at which the service exposes itself, using the `corbaloc` syntax.

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

iiop:payload

Synopsis

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

Description

The `iiop:payload` element is a child of the WSDL `port` element. It specifies the type of payload being passed through the IIOP tunnel. If the iiop:payload element is set, Artix will use the information to attempt codeset negotiation on the contents of the payload being sent through the tunnel. If you do not want codeset negotiation attempted, do not use this element in your IIOP Tunnel port definition.

Attributes

The `iiop:payload` element has a single required element named `type`. The `type` attribute specifies the type of data contained in the payload.

Examples

If your payload contains string data and you want Artix to attempt codeset negotiation you would use the following:

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

iiop:policy

Synopsis

```
<iiop:policy
poaname="..."|persistent="..."|serviceid="..." />
```

Description

The `iiop:policy` element is a child of a WSDL `port` element. It specifies the POA polices the Artix service will use when creating the POA for the IIOP port. Each `iiop:policy` element can only specify one policy. Therefore to define multiple policies you must use multiple `iiop:policy` elements.

Attributes

The `iiop:policy` element uses attributes to specify the policy it is describing. The following attributes are used:

`poaname`	Specifies the POA name to use when creating the IIOP port. The default POA name is `WS_ORB`.
`persistent`	Specifies the value of the POA's persistence policy. The default is `false`; the POA is not persistent.

`serviceid`	Specifies the value of the POA's ID. By default, Artix POAs are assigned their IDs by the ORB.

See Also

For more information about CORBA POA policies see the Orbix documentation.

WebSphere MQ Port

Artix provides a number of WSDL extensions to configure a WebSphere MQ service.

Artix Extension Elements

Runtime Compatibility

The WebSphere MQ transport's extension elements are only compatible with the C++ runtime.

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 use a WebSphere MQ port you need to include the following in the `definitions` tag of your contract:

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

mq:client

Synopsis

```
<mq:client QueueManager="..." QueueName="..."
           ReplyQueueManager="..." ReplyQueueName="..."
           Server_Client="..." ModelQueueName="..."
           AliasQueueName="..." ConnectionName="..."
           ConnectionReusable="..." ConnectionFastPath="..."
           UsageStyle="..." CorrelationStyle="..." AccessMode="..."
           Timeout="..." MessageExpiry="..." MessagePriority="..."
           Delivery="..." Transactional="..." ReportOption="..."
           Format="..." MessageID="..." CorrelationID="..."
           ApplicationData="..." AccountingToken="..."
           ApplicationIdData="..." ApplicationOriginData="..."
           UserIdentification="..." />
```

Description

The `mq:client` element is used to configure a client endpoint for connecting to WebSphere MQ. For an MQ client endpoint that receives replies you must provide values for the `QueueManager`, `QueueName`, `ReplyQueueManager`, and `ReplyQueueName` attributes. If the endpoint is not going to receive replies, you do not need to supply settings for the reply queue.

Attributes

The `mq:client` element has the following attributes:

`QueueManager`	Specifies the name of the queue manager used for making requests.
`QueueName`	Specifies the name of the queue used for making requests.
`ReplyQueueName`	Specifies the name of the queue used for receiving responses.

ReplyQueueManager	Specifies the name of the queue manager used for receiving responses.
Server_Client	Specifies which MQ libraries are to be used.
ModelQueueName	Specifies the name of the queue to use as a model for creating dynamic queues.
AliasQueueName	Specifies the local name of the reply queue when the reply queue manager is not on the same host as the client's local queue manager.
ConnectionName	Specifies the name of the connection Artix uses to connect to its queue.
ConnectionReusable	Specifies if the connection can be used by more than one application. The default is `false`; the connection is not reusable.
ConnectionFastPath	Specifies if the queue manager will be loaded in process. The default is `false`; the queue manager runs as a separate process.
UsageStyle	Specifies if messages can be queued without expecting a response.
CorrelationStyle	Specifies what identifier is used to correlate request and response messages.
AccessMode	Specifies the level of access applications have to the queue.
Timeout	Specifies the amount of time, in milliseconds, between a request and the corresponding reply before an error message is generated.
MessageExpiry	Specifies the value of the MQ message descriptor's `Expiry` field. It specifies the lifetime of a message in tenths of a second. The default value is `INFINITE`; messages never expire.
MessagePriority	Specifies the value of the MQ message descriptor's `Priority` field.
Delivery	Specifies the value of the MQ message descriptor's `Persistence` field.
Transactional	Specifies if transaction operations must be performed on the messages.
ReportOption	Specifies the value of the MQ message descriptor's `Report` field.
Format	Specifies the value of the MQ message descriptor's `Format` field.
MessageID	Specifies the value of the MQ message descriptor's `MsgId` field. A value must be specified if `CorrelationStyle` is set to `none`.
CorrelationID	Specifies the value for the MQ message descriptor's `CorrelId` field. A value must be specified if `CorrelationStyle` is set to `none`.

ApplicationData	Specifies any application-specific information that needs to be set in the message header.
AccountingToken	Specifies the value for the MQ message descriptor's AccountingToken field.
ApplicationIdData	Specifies the value for the MQ message descriptor's ApplIdentityData field.
ApplicationOriginData	Specifies the value for the MQ message descriptor's ApplOriginData field.
UserIdentification	Specifies the value for the MQ message descriptor's UserIdentifier field.

mq:server

Synopsis

```
<mq:server QueueManager="..." QueueName="..."
          ReplyQueueManager="..." ReplyQueueName="..."
          Server_Client="..." ModelQueueName="..."
          ConnectionName="..." ConnectionReusable="..."
          ConnectionFastPath="..." UsageStyle="..."
          CorrelationStyle="..." AccessMode="..." Timeout="..."
          MessageExpiry="..." MessagePriority="..." Delivery="..."
          Transactional="..." ReportOption="..." Format="..."
          MessageID="..." CorrelationID="..." ApplicationData="..."
          AccountingToken="..." ApplicationOriginData="..."
          PropogateTransactions="..." />
```

Description

The mq:server element is used to configure a server endpoint for connecting to WebSphere MQ. For an MQ server endpoint you must provide values for the QueueManager and QueueName attributes.

Attributes

The mq:server element has the following attributes:

QueueManager	Specifies the name of the queue manager used for receiving requests.
QueueName	Specifies the name of the queue used to receive requests.
ReplyQueueName	Specifies the name of the queue where responses are placed. This setting is ignored if the client specifies a ReplyToQ in a request's message descriptor.
ReplyQueueManager	Specifies the name of the reply queue manager. This setting is ignored if the client specifies a ReplyToQMgr in a request's message descriptor.
Server_Client	Specifies which MQ libraries are to be used.
ModelQueueName	Specifies the name of the queue to use as a model for creating dynamic queues.
ConnectionName	Specifies the name of the connection Artix uses to connect to its queue.

ConnectionReusable	Specifies if the connection can be used by more than one application. The default is false; the connection is not reusable.
ConnectionFastPath	Specifies if the queue manager will be loaded in process. The default is false; the queue manager runs as a separate process.
UsageStyle	Specifies if messages can be queued without expecting a response.
CorrelationStyle	Specifies what identifier is used to correlate request and response messages.
AccessMode	Specifies the level of access applications have to the queue.
Timeout	Specifies the amount of time, in milliseconds, between a request and the corresponding reply before an error message is generated.
MessageExpiry	Specifies the value of the MQ message descriptor's Expiry field. It specifies the lifetime of a message in tenths of a second. The default value is INFINITE; messages never expire.
MessagePriority	Specifies the value of the MQ message descriptor's Priority field.
Delivery	Specifies the value of the MQ message descriptor's Persistence field.
Transactional	Specifies if transaction operations must be performed on the messages.
ReportOption	Specifies the value of the MQ message descriptor's Report field.
Format	Specifies the value of the MQ message descriptor's Format field.
MessageID	Specifies the value of the MQ message descriptor's MsgId field. A value must be specified if CorrelationStyle is set to none.
CorrelationID	Specifies the value for the MQ message descriptor's CorrelId field. A value must be specified if CorrelationStyle is set to none.
ApplicationData	Specifies any application-specific information that needs to be set in the message header.
AccountingToken	Specifies the value for the MQ message decscriptor's AccountingToken field.
ApplicationOriginData	Specifies the value for the MQ message descriptor's ApplOriginData field.
PropogateTransactions	Specifies if local MQ transactions should be included in flowed transactions. Default is true.

Options

Table 10 describes the correlation between the Artix attribute settings and the MQOPEN settings.

Attribute Details

Server_Client

Description

The `Server_Client` attribute specifies which shared libraries to load on systems with a full WebSphere MQ installation.

Parameters

Table 7 describes the settings for this attribute for each type of WebSphere MQ installation.

Table 7: *Server_Client Attribute Settings*

MQ Installation	Server_Client Setting	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.
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.

AliasQueueName

Description

The `AliasQueueName` attribute specifies the local name of the reply queue when the service's queue manager is running a different host from the client. Using this attribute 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.

Effect of AliasQueueName

When you specify a value for the `AliasQueueName` attribute in an `mq:client` element, you alter how Artix populates the request's `ReplyToQ` field and `ReplyToQMgr` field. Typically, Artix populates the reply queue information in the request's message descriptor with the values specified in `ReplyQueueManager` and `ReplyQueueName`. Setting `AliasQueueName` causes Artix to leave `ReplytoQMgr` empty and to set `ReplyToQ` to the value of `AliasQueueName`. 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.

Examples

If you had a system defined similar to that shown in Figure 1, 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`.

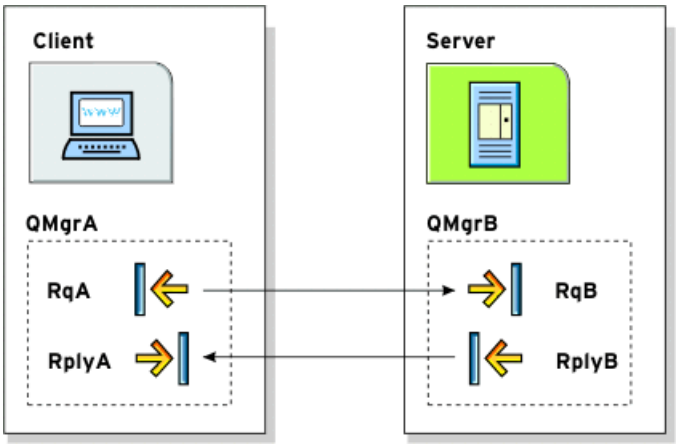

Figure 1: *MQ Remote Queues*

The Artix WebSphere MQ port definitions for the client and server for this deployment are shown in Example 42. `AliasQueueName` is set to `RplyB` because that is the remote queue proxying for the reply queue in server's local queue manager. `ReplyQueueManager` and `ReplyQueueName` 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` and `ReplyQueueName` 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 42: *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"/>
```

UsageStyle

Description

The `UsageStyle` specifies if a message can be queued without expecting a response. The default value is `Requester`.

Options

The valid settings for `UsageStyle` are described in Table 8.

Table 8: *UsageStyle Settings*

Attribute Setting	Description
`Peer`	Specifies that messages can be queued without expecting any response.
`Requester`	Specifies that the message sender expects a response message. This is the default.
`Responder`	Specifies that the response message must contain enough information to facilitate correlation of the response with the original message.

Examples

In Example 43, the WebSphere MQ client wants a response from the server and needs to be able to associate the response with the request that generated it. Setting the `UsageStyle` to responder ensures that the server's response will properly populate the response message descriptor's `CorrelID` field according to the defined correlation style. In this case, the correlation style is set to `correlationId`.

Example 43: *MQ Client with UsageStyle Set*

```
<mq:client QueueManager="postmaster" QueueName="eddie"
           ReplyQueueManager="postmaster"
   ReplyQueueName="fred"
           UsageStyle="responder"
           CorrelationStyle="correlationId"/>
```

CorrelationStyle

Description

The `CorrelationStyle` attribute specifies how WebSphere MQ matches both the message identifier and the correlation identifier to select a particular message to be retrieved from the queue (this is accomplished by setting the corresponding `MQMO_MATCH_MSG_ID` and `MQMO_MATCH_CORREL_ID` in the `MatchOptions` field in `MQGMO` to indicate that those fields should be used as selection criteria).

Options

The valid correlation styles for an Artix WebSphere MQ port are `messageId`, `correlationId`, and `messageId copy`.

Note: When a value is specified for `ConnectionName`, you cannot use `messageId copy` as the correlation style.

Table 9 shows the actions of `MQGET` and `MQPUT` when receiving a message using a WSDL specified message ID and a WSDL specified correlation ID.

Table 9: *MQGET and MQPUT Actions*

Artix Port Setting	Action for MQGET	Action for MQPUT
`messageId`	Set the `CorrelId` of the message descriptor to value of the `MessageID`.	Copy the value of the `MessageID` onto the message descriptor's `CorrelId`.
`correlationId`	Set `CorrelId` of the message descriptor to that value of the `CorrelationID`.	Copy value of the `CorrelationID` onto message descriptor's `CorrelId`.
`messageId copy`	Set `MsgId` of the message descriptor to value of the `messageID`.	Copy the value of the `MessageID` onto message descriptor's `MsgId`.

AccessMode

Description

The `AccessMode` attribute controls the action of `MQOPEN` and `MQPUT` in the Artix WebSphere MQ transport.

Table 10: *Artix WebSphere MQ Access Modes*

Attribute Setting	Description
`peek`	`peek` opens a queue to browse messages. Equivalent to `MQOO_BROWSE`. This setting is not valid for remote queues.
`send`	`send` has the same effect as `send+setall` for backward compatibility reasons.

Table 10: *Artix WebSphere MQ Access Modes*

Attribute Setting	Description
send+setall	send+setall opens a queue to put messages into. The queue is opened for use with subsequent MQPUT calls. Equivalent to: MQOPEN => MQOO_SET_ALL_CONTEXT | MQOO_OUTPUT MQPUT => MQPMO_SET_ALL_CONTEXT You can specify different authorizations using other send+ settings (for example, send+setid).
send+setid	Equivalent to: MQOPEN => MQOO_SET_IDENTITY_CONTEXT | MQOO_OUTPUT MQPUT => MQPMO_SET_IDENTITY_CONTEXT
send+passall	Equivalent to: MQOPEN => MQOO_PASS_ALL_CONTEXT | MQOO_OUTPUT MQPUT => MQPMO_PASS_ALL_CONTEXT
send+passid	Equivalent to: MQOPEN => MQOO_PASS_IDENTITY_CONTEXT | MQOO_OUTPUT MQPUT => MQPMO_PASS_IDENTITY_CONTEXT
send+none	Equivalent to MQOO_OUTPUT only. This setting has no associated authorization level.
receive (default)	receive opens a queue to get messages using a queue-defined default. Equivalent to MQOO_INPUT_AS_Q_DEF. The default value depends on the DefInputOpenOption queue attribute (MQOO_INPUT_EXCLUSIVE or MQOO_INPUT_SHARED).
receive exclusive	receive exclusive opens a queue to get messages with exclusive access. Equivalent to MQOO_INPUT_EXCLUSIVE. The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open (by this or another application) for input of any type.
receive shared	receive shared opens queue to get messages with shared access. Equivalent to MQOO_INPUT_SHARED. The queue is opened for use with subsequent MQGET calls. The call can succeed if the queue is currently open by this or another application with MQOO_INPUT_SHARED.

MessagePriority

Description

The MessagePriority attribute specifies the value for the MQ message descriptor's Priority field. Its value must be greater than or equal to zero; zero is the lowest priority. Special values for MessagePriority include highest (9), high (7), medium (5), low (3) and lowest (0). The default is normal.

Delivery

Description The `Delivery` attribute specifies the value of the MQ message descriptor's `Persistence` field.

Options Table 11 describes the settings for `Delviery`.

Table 11: *Delivery Attribute Settings*

Artix	WebSphere MQ
`persistent`	`MQPER_PERSISTENT`
`not persistent (Default)`	`MQPER_NOT_PERSISTENT`

To support transactional messaging, you must make the messages `persistent`.

Transactional

Description The `Transactional` controls how messages participate in transactions and what role WebSphere MQ plays in the transactions.

Options The values of the `Transactional` attribute are explained in Table 12.

Table 12: *Transactional Attribute Settings*

Attribute Setting	Description
`none (Default)`	The messages are not part of a transaction. No rollback actions will be taken if errors occur.
`internal`	The messages are part of a transaction with WebSphere MQ serving as the transaction manager.
`xa`	The messages are part of a flowed transaction with WebSphere MQ serving as an enlisted resource manager.

When the `transactional` attribute to `internal` for an Artix service, the following happens during request processing:

1. When a request is placed on the service's request queue, MQ begins a transaction.
2. The service processes the request.
3. Control is returned to the server transport layer.
4. If no reply is required, the local transaction is committed and the request is permanently discarded.
5. If a reply message is required, the local transaction is committed and the request is permanently discarded only after the reply is successfully placed on the reply queue.
6. If an error is encountered while the request is being processed, the local transaction is rolled back and the request is placed back onto the service's request queue.

Examples

Example 44 shows the settings for a WebSphere MQ server port whose requests will be part of transactions managed by WebSphere MQ. Note that the `Delivery` attribute must be set to `persistent` when using transactions.

Example 44: *MQ Client Setup to use Transactions*

```
<mq:server QueueManager="herman" QueueName="eddie"
           ReplyQueueManager="gomez"
   ReplyQueueName="lurch"
           UsageStyle="responder" Delivery="persistent"
           CorrelationStyle="correlationId"
           Transactional="internal"/>
```

ReportOption

Description

The `ReportOption` attribute is mapped to the MQ message descriptor's `Report` field. It enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and how the message and correlation identifiers in the report or reply message are to be set. Artix only allows you to specify one `ReportOption` per Artix port. Setting more than one will result in unpredictable behavior.

Options

The values of this attribute are explained in Table 13.

Table 13: *ReportOption Attribute Settings*

Attribute Setting	Description
`none (Default)`	Corresponds to `MQRO_NONE`. `none` specifies that no reports are required. You should never specifically set `ReportOption` to `none`; it will create validation errors in the contract.
`coa`	Corresponds to `MQRO_COA`. `coa` specifies that confirm-on-arrival reports are required. This type of report is generated by the queue manager that owns the destination queue, when the message is placed on the destination queue.
`cod`	Corresponds to `MQRO_COD`. `cod` specifies that confirm-on-delivery reports are required. This type of report is generated by the queue manager when an application retrieves the message from the destination queue in a way that causes the message to be deleted from the queue.

Table 13: *ReportOption Attribute Settings*

Attribute Setting	Description
`exception`	Corresponds to `MQRO_EXCEPTION`. `exception` specifies that exception reports are required. This type of report can be generated by a message channel agent when a message is sent to another queue manager and the message cannot be delivered to the specified destination queue. For example, the destination queue or an intermediate transmission queue might be full, or the message might be too big for the queue.
`expiration`	Corresponds to `MQRO_EXPIRATION`. `expiration` specifies that expiration reports are required. This type of report is generated by the queue manager if the message is discarded prior to delivery to an application because its expiration time has passed.
`discard`	Corresponds to `MQRO_DISCARD_MSG`. `discard` indicates that the message should be discarded if it cannot be delivered to the destination queue. An exception report message is generated if one was requested by the sender

Format

Description

The `Format` attribute is mapped to the MQ message descriptor's `Format` field. It specifies an optional format name to indicate to the receiver the nature of the data in the message.

Options

The value may contain any character in the queue manager's character set, but it is recommended that the name be restricted to the following:

- Uppercase A through Z
- Numeric digits 0 through 9

In addition, the `FormatType` attribute can take the special values `none`, `string`, `event`, `programmable command`, and `unicode`. These settings are described in Table 14.

Table 14: *FormatType Attribute Settings*

Attribute Setting	Description
`none (Default)`	Corresponds to `MQFMT_NONE`. No format name is specified.

Table 14: *FormatType Attribute Settings*

Attribute Setting	Description
`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 an 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.

When you are interoperating with WebSphere MQ applications hosted on a mainframe and the data needs to be converted into the systems native data format, you should set `Format` to `string`. Not doing so will result in the mainframe receiving corrupted data.

Tuxedo Port

Artix can connect to applications that use BEA's Tuxedo as their messaging backbone.

Runtime Compatibility

The Tuxedo transport's extension elements are only compatible with the C++ runtime.

Namespace

The extensions used to describe a Tuxedo port are defined in the namespace `http://schemas.iona.com/transports/tuxedo`. When a Tuxedo endpoint is defined in a contract, the contract will need the following namespace declaration in the contract's `definition` element:

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

tuxedo:server

Synopsis

```
<tuxedo:server>
  <tuxedo:service ...>
    ...
  </tuxedo:service>
</tuxedo:server>
```

Description

The `tuxedo:server` element is a child of a WSDL `port` element. It contains the definition of a Tuxedo endpoint.

tuxedo:service

Synopsis

```
<tuxedo:service name="...">
   <tuxedo:input .../>
   ...
</tuxedo:service>
```

Description

The `tuxedo:service` element is the child of a tuxedo:server element. It specifies the bulletin board name used to post and receive messages. It has a number of tuxedo:input child elements that provide a map to the operations from which messages are routed.

Attributes

The tuxedo:service element has a single required attribute called `name`. The `name` attribute specifies the bulletin board name for the service.

tuxedo:input

Synopsis

```
<tuxedo:input operation="..." />
```

Description

The `tuxedo:input` element specify which of the operations bound to the port being defined are handled by the Tuxedo service.

Attributes

The `tuxedo:input` element has a single required attribute called `operation`. The `operation` attribute specifies the WSDL operation that

is handled by the Tuxedo service. The value must correspond the value of the `name` attribute of the appropriate WSDL `operation` element.

JMS Port

JMS is a powerful messaging system used by Java applications.

C++ Runtime Extensions

Namespace

The WSDL extensions used to describe JMS transport details for the C++ runtime are defined in the namespace `http://celtix.objectweb.org/transports/jms`. If you are going to use a JMS port you need to include the following in the `definitions` tag of your contract:

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

jms:address

Synopsis

```
<jms:address destinationStyle="..."
             jndiConnectionFactoryName="..."
             jndiDestinationName="..."
             jndiReplyDestinationName="..."
             jmsDestinationName="..."
             jmsReplyDestinationName="..."
             connectionUserName="..." connectionPassword="...">
  <jms:JMSNamingProperty ... />
  ...
</jms:address>
```

Description

The `jms:address` element specifies the information needed to connect to a JMS system.

Attributes

The `jms:address` element has the following attributes:

`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which Artix connects.
`jndiReplyDestinationName`	Specifies the JNDI name bound to the JMS destination where replies are sent. This attribute allows you to use a user defined destination for replies.
`jmsDestinationName`	Specifies the JMS name of the JMS destination to which requests are sent.
`jmsReplyDestinationName`	Specifies the JMS name of the JMS destination where replies are sent. This attribute allows you to use a user defined destination for replies.

connectionUserName	Specifies the username to use when connecting to a JMS broker.
connectionPassword	Specifies the password to use when connecting to a JMS broker.

jms:JMSNamingProperty

Synopsis

```
<jms:JMSNamingProperty name="..." value="..." />
```

Description

The jms:JMSNamingProperty element is a child of the jms:address element. It is used to provide the values used to populate the properties object used when connecting to a JNDI provider.

Attributes

The jms:JMSNamingProperty element has the following attributes:

name	Specifies the name of the JNDI property to set.
value	Specifies the value for the specified property.

JNDI property names

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.

jms:client

Synopsis

```
<jms:client messageType="..." />
```

Description

The jms:client element is a child of the WSDL port element. It is used to specify the types of messages being used by a JMS client endpoint and the timeout value for a JMS client endpoint.

Attributes

The jms:client element has the following attributes:

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.

jms:server

Synopsis

```
<jms:server useMessageIDAsCorrelationID="..."
            durableSubscriberName="..."
            messageSelector="..." transactional="..." />
```

Description

The `jms:server` element is a child of the WSDL `port` element. It specifies settings used to configure the behavior of a JMS service endpoint.

Attributes

The `jms:server` element has the following attributes:

`useMessageIDAsCorrealationID`	Specifies whether JMS will use the message ID to correlate messages. The default is `false`.
`durableSubscriberName`	Specifies the name used to register a durable subscription.
`messageSelector`	Specifies the string value of a message selector to use.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false`.

File Transfer Protocol Port

Artix can use an FTP server as a middle-tier message broker.

Runtime Compatibility

The FTP transport's extensions are compatible with the C++ runtime.

Namespace

The extensions used to describe a File Transfer Protocol (FTP) port are defined in the namespace `http://schemas.iona.com/transports/ftp`. When an FTP endpoint is defined in a contract, the contract will need the following namespace declaration in the contract's `definition` element:

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

ftp:port

Synopsis

```
<ftp:port host="..." port="..." requestLocation="..."
          replyLocation="..." connectMode="..." scanInterval="...">
  <ftp:properties>
    ...
  </ftp:properties>
</ftp:port>
```

Description

The `ftp:port` element is a child of a WSDL `port` element. It defines the connection details for an FTP endpoint. It may contain an ftp:properties element.

Attributes

The `ftp:port` element has the following attributes:

`host`	Specifies the domain name or IP address of the machine hosting the FTPD used by the endpoint.
`port`	Specifies the port number on which the endpoint will contact the FTPD.
`requestLocation`	Specifies the path on the FTPD host the endpoint will use for requests. The default is `/`.
`replyLocation`	Specifies the path on the FTPD host the endpoint will use for replies. The default is `/`.
`connectMode`	Specifies the connection mode used to connect to the FTPD. Valid values are `passive` and `active`. The default is `passive`.
`scanInterval`	Specifies the interval, in seconds, at which the request and reply directories are scanned for updates. The default is `5`.

ftp:properties

Synopsis

```
<ftp:properties>
  <ftp:property ... />
  ...
</ftp:property>
```

Description

The `ftp:properties` element defines a number of file naming properties used by the endpoint for storing requests and replies. It contains one or more ftp:property elements.

ftp:property

Synopsis

```
<ftp:property name="..." value="..." />
```

Description

The `ftp:property` element defines specific file naming properties to use when reading and writing messages on the FTPD host. The properties are defined by the implementation used for the naming scheme classes. Artix provides a default implementation. However, a custom naming scheme implementation may have different properties.

Attributes

The `ftp:property` element has the following attributes:

`name`	Specifies the name of the property to set.
`value`	Specifies the value of the property.

Default Naming Properties

The default naming implementation provided with Artix supports the following properties:

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

Part III

Other Extensions

In this part

This part contains the following chapters:

Routing

Artix provides a number of WSDL extensions for defining how messages are routed between services.

Runtime Compatibility

The extensions described below are only recognized by the Artix router.

Namespace

The Artix routing elements are defined in the `http://schemas.iona.com/routing` namespace. When describing routes in an Artix contract your contract's `definition` element must have the following entry:

```
xmlns:routing="http://schemas.iona.com/routing"
```

routing:expression

Synopsis

```
<routing:expression name="..." evaluator="..."
  ...
</routing:expression>
```

Description

The `routing:expression` element is a child of the WSDL `definitions` element. It specifies an XPATH expression that evaluates messages for content-based routing.

Attributes

The `routing:expression` requires the following two attributes:

`name`	Specifies a string that is used to refer to the expression when defining routes.
`evaluator`	Specifies the name of the grammar used in the expression. Currently the only valid value is `xpath`.

routing:route

Synopsis

```
<routing:route name="..." mulitRoute="...">
  ...
</routing:route>
```

Description

The `routing:route` element is the root element of each route described in a contract.

Attributes

The `routing:route` element takes the following attributes:

`name`	Specifies a unique identifier for the route. This attribute is required.
`multiRoute`	An optional attribute that specifies how messages are sent to the listed destinations. Values are `fanout`, `failover`, or `loadBalance`. Default is to route messages to a single destination.

Options

Standard routes define a single source/destination pair. When the `mulitRoute` attribute is specified, your route description will contain more than one destination.

Setting the `multiRoute` attribute has the following effects:

- `fanout` instructs Artix to send messages from the source to all the listed destinations.
- `failover` instructs Artix to move through the list of destinations until it can successfully send the message.
- `loadBalance` instructs Artix to use a round-robin algorithm to spread messages across all of the listed destinations.

routing:source

Synopsis

```
<routing:source service="..." port="..." />
```

Description

The `routing:source` element is a child of a routing:route element. It specifies the port from which the route will redirect messages. A route can have several source elements as long as they all meet the compatibility rules for port-based routing.

Attributes

The `routing:source` element requires two attributes:

`service`	Specifies the WSDL `service` element in which the source port is defined.
`port`	Specifies the name of the WSDL `port` element from which messages are being received. The router will create a proxy to listen for messages on this port.

routing:query

Synopsis

```
<routing:query expression="...">
  <routing:desitination id="..." ... />
  ...
</routing:query>
```

Description

The `routing:query` element is a child of a routing:route element. It specifies the destinations for a content-based route. The child routing:destination elements must use the `id` attribute to specify the value used to select the destination.

Attributes

The `routing:query` element has one attribute:

`expression`	Specifies the value of the `name` attribute from the routing:expression element defining the XPATH expression used to select the destination of the message. The query selects the destination with the id value that matches the result of applying the expression to the message content.

routing:destination

Synopsis

```
<routing:destination value="..." service="..."
                    port="..." route="..." />
```

Description

The `routing:destination` element is a child of a `routing:route` element. It specifies the port to which the source messages are directed. The destination must be compatible with all of the source elements.

Attributes

The `routing:destination` element has the following attributes:

`value`	Specifies the value of the content-based routing query that triggers the destination. This attribute is required when the element is the child of a routing:query element and ignored otherwise.
`service`	Specifies the WSDL `service` element in which the destination port is defined.
`port`	Specifies the name of the port WSDL `element` to which messages are routed.
`route`	Specifies a linked route to use for selecting the ultimate destination. When this attribute is used, you should not use the `service` attribute or the `port` attribute.

routing:transportAttribute

Synopsis

```
<routing:transportAttribute>
  ...
</routing:transportAttribute>
```

Description

The `routing:transportAttribute` element is a child of a routing:route element. It defines routing rules based on the transport attributes set in a message's header when using HTTP, CORBA, or WebSphere MQ. The criteria for determining if a message meets the transport attribute rule are specified using the following child elements:

- routing:equals
- routing:greater
- routing:less
- routing:startswith
- routing:endswith
- routing:contains
- routing:empty
- routing:nonempty

A message passes the rule if it meets each criterion specified by the child elements.

Transport attribute rules are defined after all of the operation-based routing rules and before any destinations are listed.

Examples

Example 45 shows a route using transport attribute rules based on HTTP header attributes. Only messages sent to the server whose

UserName is equal to JohnQ will be passed through to the destination port.

Example 45: *Transport Attribute Rules*

```
<routing:route name="httpTransportRoute">
  <routing:source service="tns:httpService"
                  port="tns:httpPort"/>
  <routing:trasnportAttributes>
    <rotuing:equals
              contextName="http-conf:HTTPServerIncomingContexts"
               contextAttributeName="UserName"
               value="JohnQ"/>
  </routing:transportAttributes>
  <routing:destination service="tns:httpDest"
                       port="tns:httpDestPort"/>
</routing:route>
```

routing:equals

Synopsis

```
<routing:equals contextName="..."
                contextAttributeName="..."
                value="..."
                ingnorecase="..." />
```

Description

The routing:equals element is a child of a routing:transportAttribute element. It defines a rule that is triggered when the specified attribute equals the value given. It applies to string or numeric attributes.

Attributes

The routing:equals element has the following attributes:

contextName	Specifies the QName of the context in which the desired transport attributes are stored.
contextAttributeName	Specifies the QName of the transport attribute the rule evaluates.
value	Specifies the value against which the specified attribute is evaluated.
ignorecase	Specifies whether the case of characters in a string are ignored. The default is no; case is considered when evaluating string data.

routing:greater

Synopsis

```
<routing:greater contextName="..."
                 contextAttributeName="..."
                 value="..." />
```

Description

The routing:greater element is a child of a routing:transportAttribute element. It defines a rule that is triggered when the value of the specified attribute is greater than the value given. It applies to numeric attributes.

Attributes

The `routing:greater` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.
`value`	Specifies the value against which the specified attribute is evaluated.

routing:less

Synopsis

```
<routing:less contextName="..."
              contextAttributeName="..."
              value="..." />
```

Description

The `routing:less` element is a child of a routing:transportAttribute element. It defines a rule that is triggered when the value of the specified attribute is less than the value given. It applies to numeric attributes.

Attributes

The `routing:less` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.
`value`	Specifies the value against which the specified attribute is evaluated.

routing:startswith

Synopsis

```
<routing:startswith contextName="..."
                    contextAttributeName="..."
                    value="..."
                    ingnorecase="..." />
```

Description

The `routing:startswith` element is a child of a routing:transportAttribute element. It applies to string attributes and tests whether the attribute starts with the specified value.

Attributes

The `routing:startswith` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.
`value`	Specifies the value against which the specified attribute is evaluated.
`ignorecase`	Specifies whether the case of characters in a string are ignored. The default is `no`; case is considered when evaluating string data.

routing:endswith

Synopsis

```
<routing:endswith contextName="..."
                  contextAttributeName="..."
                  value="..."
                  ingnorecase="..." />
```

Description

The `routing:endswith` element is a child of a routing:transportAttribute element. It applies to string attributes and tests whether the attribute ends with the specified value.

Attributes

The `routing:endswith` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.
`value`	Specifies the value against which the specified attribute is evaluated.
`ignorecase`	Specifies whether the case of characters in a string are ignored. The default is `no`; case is considered when evaluating string data.

routing:contains

Synopsis

```
<routing:contains contextName="..."
                  contextAttributeName="..."
                  value="..."
                  ingnorecase="..." />
```

Description

The `routing:contains` element is a child of a routing:transportAttribute element. It applies to string or list attributes. For strings, it tests whether the attribute contains the value. For lists, it tests whether the value is a member of the list.

Attributes

The `routing:contains` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.
`value`	Specifies the value against which the specified attribute is evaluated.
`ignorecase`	Specifies whether the case of characters in a string are ignored. The default is `no`; case is considered when evaluating string data.

routing:empty

Synopsis

```
<routing:empty contextName="..."
               contextAttributeName="..." />
```

Description

The `routing:empty` element is a child of a routing:transportAttribute element. It applies to string or list attributes. For lists, it tests whether the list is empty. For strings, it tests for an empty string.

Attributes

The `routing:empty` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.

routing:nonempty

Synopsis

```
<routing:nonempty contextName="..."
                  contextAttributeName="..." />
```

Description

The `routing:nonempty` element is a child of a routing:transportAttribute element. It applies to string or list attributes. For lists, it passes if the list is not empty. For strings, it passes if the string is not empty.

Attributes

The `routing:nonempty` element has the following attributes:

`contextName`	Specifies the QName of the context in which the desired transport attributes are stored.
`contextAttributeName`	Specifies the QName of the transport attribute the rule evaluates.

Transport Attribute Context Names

The `contextName` attribute is specified using the QName of the context in which the attribute is defined. The contexts shipped with Artix are described in Table 15.

Table 15: *Context QNames*

Context QName	Details
`http-conf:HTTPServerIncomingContexts`	Contains the attributes for HTTP messages being received by a server.
`corba:corba_input_attributes`	Contains the data stored in the CORBA principle
`mq:MQConnectionAttributes`	Contains the attributes used to connect to an MQ queue.

Table 15: *Context QNames*

Context QName	Details
`mq:MQIncomingMessageAttributes`	Contains the attributes in the message header of an MQ message.
`bus-security`	Contains the attributes used by the security service to secure your services.

Security

Artix uses a special WSDL extension element to specify security policies for endpoints.

Runtime Compatibility

The security extensions are only compatible with C++ runtime.

Namespace

The elements Artix uses for specifying security policies are defined in the `http://schemas.iona.com/bus/security` namespace. When defining security policies in an Artix contract your contract's `definition` element must have the following entry:

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

bus-security:security

Synopsis

```
<bus-security:security enableSecurity="..."
                       is2AuthorizationActionRoleMapping="..."
                       enableAuthorization="..."
                       authenticationCacheSize="..."
                       authenticationCacheTimeout ="..."
                       securityType="..."
                       securityLevel="..."
                       authorizationRealm="..."
                       defaultPassword="..." />
```

Description

The `bus-security:security` element is a child of a WSDL `port` element. It's attributes specify security policies for the endpoint.

Attributes

The `bus-security:security` element has the following attributes:

`enableSecurity`	Specifies if the service should loud the ASP plug-in. Default is `false`.
`is2AuthorizationActionRoleMapping`	Specifies the URL of the action role mapping file the Artix security framework uses to authenticate requests for this endpoint.
`enableAuthorization`	Specifies if the endpoint should use the Artix security framework for authentication. Default is `false`.
`enableSSO`	Specifies if the service can use single-sign on (SSO). Default is `false`.

`authenticationCacheSize`	Specifies the maximum number of credentials stored in the authentication cache. A value of `-1` (the default) means unlimited size. A value of `0` disables the cache.
`authenticationCacheTimeout`	Specifies the time (in seconds) after which a credential is considered stale. A value of `-1` (the default) means an infinite time-out. A value of `0` disables the cache.
`securityLevel`	Specifies the level from which security credentials are picked up. The following options are supported by the Artix security framework: • `MESSAGE_LEVEL`—Get security information from the transport header. This is the default. • `REQUEST_LEVEL`—Get the security information from the message header.
`authorizationRealm`	Specifies the Artix authorization realm to which an Artix server belongs. The value of this variable determines which of a user's roles are considered when making an access control decision.The default is `IONAGlobalRealm`.
`defaultPassword`	Specifies the password to use on the server side when the client credentials originate either from a CORBA Principal (embedded in a SOAP header) or from a certificate subject. The default is `default_password`.

See Also

For more information about Artix security policies, see the ***Artix Security Guide***.

Codeset Conversion

For transports that do not natively support codeset conversion Artix has the ability to perform codeset conversion.

Runtime Compatibility

The extension elements used to configure codeset conversion are only compatible with the C++ runtime.

Namespace

The elements Artix uses for defining codeset conversion rules are defined in the `http://schemas.iona.com/bus/i18n/context` namespace. When defining codeset conversion rules in an Artix contract your contract's `definition` element must have the following entry:

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

i18n-context:client

Synopsis

```
<i18n-context:client LocalCodeSet="..." OutboundCodeSet="..."
                     InboundCodeSet="..." />
```

Description

The `i18n-context:client` element is a child of a WSDL `port` element. It specifies codeset conversion rules for Artix endpoints that are acting as servers.

Attributes

The `i18n-context:client` element has the following attributes for defining how message codesets are converted:

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

i18n-context:server

Synopsis

```
<i18n-context:server LocalCodeSet="..." OutboundCodeSet="..."
                     InboundCodeSet="..." />
```

Description

The `i18n-context:server` element is a child of a WSDL `port` element. It specifies codeset conversion rules for Artix endpoints that are acting as servers.

Attributes

The `i18n-context:server` element has the following attributes for defining how message codesets are converted:

`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 replies are converted. Default is the codeset specified in `InboundCodeSet`.
`InboundCodeSet`	Specifies the codeset into which requests are converted. Default is the codeset specified in `LocalCodeSet`.

Index

E

F

H

I

J

L

M

P

R

S

T

U

W

X