Progress® | Artix®

PROGRESS®
ARTIX™

WSDL Extension Reference

Version 5.6, August 2011

BUSINESS MAKING PROGRESS™ PROGRESS software

Updated: August 9, 2011

Contents

Preface 7

What is Covered in this Book 7

Who Should Read this Book 7

How to Use this Book 7

The Artix Documentation Library 7

Part I Bindings

SOAP 1.1 Binding 11

Runtime Compatibility 11

soap:binding 11

soap:operation 12

soap:body 13

soap:header 15

soap:fault 16

SOAP 1.2 Binding 19

Runtime Compatibility 19

wsoap12:binding 19

wsoap12:operation 20

wsoap12:body 21

wsoap12:header 23

wsoap12:fault 24

MIME Multipart/Related Binding 27

Runtime Compatibility 27

Namespace 27

mime:multipartRelated 28

mime:part 28

mime:content 28

CORBA Binding and Type Map 31

CORBA Binding Extension Elements 32

Runtime Namespace 32

corba:binding 34

corba:operation 34

corba:param 35

corba:return 35

corba:raises 36

Type Map Extension Elements 37

corba:typeMapping 37

corba:struct 38

corba:member 38

corba:enum 39

corba:enumerator 40

corba:fixed 40

corba:union 42

corba:unionbranch 42

corba:case 43

corba:alias 44

corba:array 45

corba:sequence 46

corba:exception 47

corba:anonsequence 48

corba:anonstring 50

corba:object 51

Chapter 5 XML Binding 57

Namespace 57

xformat:binding 57

xformat:body 58

Part II Ports

HTTP Port 61

Standard WSDL Elements 62

http:address 62

soap:address **62**
wsoap12:address **62**
Configuration Extensions **63**
Namespace **63**
http-conf:client **63**
http-conf:server **65**
Attribute Details **67**
AuthorizationType **67**
Authorization **67**
Accept **67**
AcceptLanguage **68**
AcceptEncoding **69**
ContentType **69**
ContentEncoding **70**
Host **70**
Connection **71**
CacheControl **71**
BrowserType **74**
Referer **74**
ProxyServer **74**
ProxyAuthorizationType **75**
ProxyAuthorization **75**
UseSecureSockets **75**
RedirectURL **76**
ServerCertificateChain **76**

CORBA Port **77**
Java Runtime Namespace **77**
corba:address **77**
corba:policy **78**

JMS Port **79**
Java Runtime Extensions **80**
Namespace **80**
jms:address **80**
jms:JMSNamingProperties **81**
jms:client **82**
jms:server **82**

Index **85**

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:

- "Bindings"—contains descriptions for all the WSDL extensions used to define the payload formats supported by Artix.
- "Ports"—contains descriptions for all the WSDL extensions used to define the transports supported by Artix.

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.

Part I

Bindings

In this part

This part contains the following chapters:

SOAP 1.1 Binding	page 11
SOAP 1.2 Binding	page 19
MIME Multipart/Related Binding	page 27
CORBA Binding and Type Map	page 31
XML Binding	page 57

CHAPTER 1

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 WDL 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 11 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 12, 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 12, 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 14 for more details.
`encodingStyle`	Used in the same way as the `encodingStyle` attribute within the `soap:body` element. See "encodingStyle" on page 15 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 14 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 15 for more details.

CHAPTER 2

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 WDL 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 20, 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

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 20, 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

CHAPTER 3

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* (ftp://ftp.isi.edu/in-notes/rfc2045.txt)
- *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* (ftp://ftp.isi.edu/in-notes/rfc2046.txt).

CHAPTER 4

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.

In this chapter

This chapter discusses the following topics:

CORBA Binding Extension Elements	page 32
Type Map Extension Elements	page 37

CORBA Binding Extension Elements

Runtime Namespace

The WSDL extensions used for the Java runtime CORBA binding and the CORBA data mappings are defined in the namespace `http://schemas.apache.org/yoko/bindings/corba`. TPrimitive 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 Java Type
`Any`	`xsd:anyType`	`corba:any`	*Java runtime -* `java.lang.Object`
`boolean`	`xsd:boolean`	`corba:boolean`	`boolean`
`char`	`xsd:byte`	`corba:char`	`byte`
`wchar`	`xsd:string`	`corba:wchar`	`java.lang.String`
`double`	`xsd:double`	`corba:double`	`double`
`float`	`xsd:float`	`corba:float`	`float`
`octet`	`xsd:unsignedByte`	`corba:octet`	`short`
`long`	`xsd:int`	`corba:long`	`int`
`long long`	`xsd:long`	`corba:longlong`	`long`
`short`	`xsd:short`	`corba:short`	`short`
`string`	`xsd:string`	`corba:string`	`java.lang.String`
`wstring`	`xsd:string`	`corba:wstring`	`java.lang.String`
`unsigned short`	`xsd:unsignedShort`	`corba:ushort`	`int`
`unsigned long`	`xsd:unsignedInt`	`corba:ulong`	`long`

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

IDL Type	XML Schema Type	CORBA Binding Type	Artix Java Type
`unsigned long long`	`xsd:unsignedLong`	`corba:ulonglong`	`java.math.BigInteger`
`Object`	`wsa:EndpointReferenceType`	`corba:object`	*Java runtime -* `org.apache.cxf.ws.addressing.EndpointReferenceType`
`TimeBase::UtcT`	`xsd:dateTime`[a]	`corba:dateTime`	`java.util.Calendar`

a. The mapping between `xsd:dateTime` and `TimeBase:UtcT` is only partial. For the restrictions see "Unsupported time/date values" on page 33

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 37.
`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 39, `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/typem
   ap">
...
  <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 be described

Example 5: *myFixed Fixed Type*

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

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/typemap"
   >
...
  <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 described as shown

Example 10: *myLong IDL*

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

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/typem
   ap">
    <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.

Attributes

`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>
```

Example 23: *Reference Sample (Continued)*

```
  <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>
    <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/ty
   pemap/">
...
  <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);
};
interface Bank
{
  ::Account find_account(in string account_id)
    raises(::AccountNotFoundException);
  ::Account create_account(in string account_id,
                           in float initial_balance)
    raises(::AccountAlreadyExistsException);
};
```

XML Binding

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

Namespace

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

Part II

Ports

In this part

This part contains the following chapters:

HTTP Port	page 61
CORBA Port	page 77
JMS Port	page 79

CHAPTER 7

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.

In this chapter

This chapter discusses the following topics:

Standard WSDL Elements	page 62
Configuration Extensions	page 63
Attribute Details	page 67

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

Namespace

Example 26 shows the namespace entries you need to add to the `definitions` element of your contract to use the Java runtime's HTTP extensions.

Example 26: *Artix Java Runtime HTTP Extension Namespaces*

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

http-conf:client

Synopsis

```
<http-conf:client ConnectionTimeout="..." RecieveTimeout="..."
                  AutoRedirect="..." MaxRetransmits="..."
                  AllowChunking="..." Accept="..."
                  AcceptLanguage="..." AcceptEncoding="..."
                  ContentType="..." Host="..." Connection="..."
                 CacheControl="..." Cookie="..." BrowserType="..."
                  Referer="..." DecoupledEndpoint="..."
                  ProxyServer="..." ProxyServerPort="..."
                  ProxyServerType="..." />
```

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:

`ConnectionTimeout`	Specifies the length of time, in milliseconds, the client tries to establish a connection before timing 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.
`AllowChunking`	Specifies whether the consumer will send requests using chunking. The default is `true`.
`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.
`AuthorizationType`	Specifies the name of the authorization scheme the client wishes to use.
`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.
`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.
`DecoupledEndpoint`	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate connection.

ProxyServer	Specifies the URL of the proxy server, if one exists along the message path.
ProxyServerPort	Specifies the port number of the proxy server.
ProxyServerType	Specifies the type of proxy server to use. The default is HTTP.

http-conf:server

Synopsis

```
<http-conf:server RecieveTimeout="..."
                  SuppressClientSendErrors="..."
                  SuppressClientReceiveErrors="..."
                  HonorKeepAlive="..." RedirectURL="..."
                  CacheControl="..." ContentLocation="..."
                  ContentType="..." ContentEncoding="..."
                  ServerType="..."
```

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:

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`.

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

Table 3: *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.

Table 3: *Settings for http-conf:client CacheControl*

Directive	Behavior
`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 4 shows the valid values for `CacheControl` in `http-conf:server`.

Table 4: *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.

Table 4: *Settings for http-conf:server CacheControl (Continued)*

Directive	Behavior
`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 4: *Settings for http-conf:server CacheControl (Continued)*

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.

CHAPTER 8

CORBA Port

Artix supports a robust mechanism for configuring a CORBA endpoint.

Java Runtime Namespace

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

```
xmlns:corba="http://schemas.apache.org/yoko/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.`

CHAPTER 9

JMS Port

JMS is a powerful messaging system used by Java applications.

In this chapter

This chapter discusses the following topic:

Java Runtime Extensions page 80

Java Runtime Extensions

Namespace

The WSDL extensions for defining a JMS endpoint are defined in the namespace `http://cxf.apache.org/transports/jms`. In order to use the JMS extensions you will need to add the line shown in Example 27 to the definitions element of your contract.

Example 27: *Java Runtime Namespace*

```
xmlns:jms="http://cxf.apache.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 used for requests.
`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:JMSNamingProperties

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`. Currently this feature is not supported by the Java runtime.

Index

A

adding a SOAP header 15, 23
arrays
  mapping to CORBA 45
Artix reference
  mapping to CORBA 51

C

complex types
  mapping to CORBA 38
corba:address 77
  location attribute 77
corba:alias 44
  name attribute 44
  repositoryID attribute 44
  type attribute 44
corba:anonsequence 48
  bound attribute 48
  elemtype attribute 49
  name attribute 48
  type attribute 49
corba:array 45
  bound attribute 46
  elemtype attribute 46
  name attribute 45
  repositoryID attribute 45
  type attribute 45
corba:binding 34
  bases attribute 34
  repositoryID attribute 34
corba:case 43
  label attribute 43
corba:enumerator 40
corba:exception 47
  name attribute 47
  repositoryID attribute 47
  type attribute 47
corba:fixed 41
  digits attribute 41
  name attribute 41
  repositoryID attribute 41
  scale attribute 41
  type attribute 41
corba:member 38
  idltype attribute 38
  name attribute 38
corba:object
  binding attribute 52
  name attribute 52
  repositoryID attribute 52
  type attribute 52
corba:operation 34
  name attribute 35
corba:param 35
  idltype attribute 35
  mode attribute 35
  name attribute 35
corba:policy 78
  persistent attribute 78
  poaname attribute 78
  serviceid attribute 78
corba:raises 36
  exception attribute 36
corba:return 35
  idltype attribute 36
  name attribute 36
corba:sequence 46
  bound attribute 46
  elemtype attribute 46
  name attribute 46
  repositoryID attribute 46
corba:typeMapping 37
  targetNamespace attribute 37
corba:union 42
  discriminator attribute 42
  name attribute 42
  repositoryID attribute 42
  type attribute 42
corba:unionbranch 43
  default attribute 43
  idltype attribute 43
  name attribute 43

D

durable subscriptions 82

E

enumerations
  mapping to CORBA 39
exceptions
  mapping to CORBA 36, 47
  mapping to SOAP 16, 24

H

http:address 62
  location attribute 62
http-conf:client 63
  Accept attribute 67
  AcceptEncoding attribute 69
  AcceptLanguage attribute 68
  AllowChunking attribute 64
  Authorization attribute 67
  AuthorizationType attribute 67
  AutoRedirect attribute 64
  BrowserType attribute 74
  CacheControl attribute 71
    cache-extension directive 72
    max-age directive 71
    max-stale directive 72
    min-fresh directive 72
    no-cache directive 71
    no-store directive 71
    no-transform directive 72
    only-if-cached directive 72
  Connection attribute 71
  ConnectionTimeout attribute 63
  ContentType attribute 64
  Cookie attribute 64
  DecoupledEndpoint attribute 64
  Host attribute 70
  ProxyAuthorization attribute 75
  ProxyAuthorizationType attribute 75
  ProxyServer attribute 74
  ReceiveTimeout attribute 63
  Referer attribute 74
  UseSecureSockets attribute 75
http-conf:server 65
  CacheControl attribute 71
    cache-extension directive 74
    max-age directive 73
    must-revalidate directive 73
    no-cache directive 72
    no-store directive 73
    no-transform directive 73
    private directive 73
    proxy-revelidate directive 73
    public directive 73
    s-maxage directive 73
  ContentEncoding attribute 70
  ContentLocation attribute 66
  ContentType attribute 66
  HonorKeepAlive attribute 66
  ReceiveTimeout attribute 65
  RedirectURL attribute 76
  ServerCertificateChain 76
  ServerType attribute 66
  SuppressClientReceiveErrors attribute 65
  SuppressClientSendErrors attribute 65
  UseSecureSockets attribute 75

I

IDL types
  fixed 40
  Object 51
  sequence 46
  typedef 44
IOR 77

J

jms:address 80
  connectionPassword attribute 81
  connectionUserName attribute 81
  destinationStyle attribute 80
  jmsDestinationName attribute 81
  jmsReplyDestinationName 81
  jndiConnectionFactoryName attribute 80
  jndiDestinationName attribute 81
  jndiReplyDestinationName 81
jms:client 82
  messageType attribute 82
jms:JMSNamingProperty 81
  name attribute 81
  value attribute 81
jms:server 82
  durableSubscriberName attribute 82
  messageSelector attribute 83
  transactional attribute 83
  useMessageIDAsCorrealationID attribute 82
JNDI
  connection factory 80

M
mime:content 28
  part attribute 29
  type attribute 29
mime:multipartRelated 28
mime:part 28
  name attribute 28

P
POA policies 78
port address
  HTTP 62
primitive types
  mapping to CORBA 32

S
soap:address 62
  location attribute 62
soap:binding 11
  style attribute 11
  transport attribute 12
soap:body 13
  encodingStyle attribute 15
  namespace attribute 15
  parts attribute 15
  use attribute 14
    encoded 14
    literal 14
soap:fault 16
  name attribute 16
  use attribute 16
    encoded 14
    literal 14
soap:header 15
  encodingStyle attribute 16
  message attribute 15
  namespace attribute 16
  part attribute 16
  use attribute 16, 24
    encoded 14
    literal 14
soap:operation 12
  soapAction attribute 13
  style attribute 13

T
timeouts
  HTTP 63

U
unions
  mapping to CORBA 42

W
wsoap12:address 62
  location attribute 62
wsoap12:binding 19
  style attribute 19
  transport attribute 20
wsoap12:body 21
  encodingStyle attribute 23
  namespace attribute 23
  parts attribute 23
  use attribute 22
    literal 22
wsoap12:fault 24
  name attribute 24
  namespace attribute 24
  use attribute 24
    literal 22
wsoap12:header 23
  encodingStyle attribute 24
  message attribute 23
  namespace attribute 24
  part attribute 23
  use attribute
    literal 22
wsoap12:operation 21
  soapAction attribute 21
  soapActionRequired attribute 21
  style attribute 21
wsoap12/
  fault
    encodingStyle attribute 25

X
xformat:binding 57
  rootNode attribute 57
xformat:body 58
  rootNode attribute 58