



Artix™

Understanding Artix Contracts

Version 4.0, March 2006

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, Orbix Mainframe, Orbix Connect, Artix, Artix Mainframe, Artix Mainframe Developer, Mobile Orchestrator, Orbix/E, Orbacus, Enterprise Integrator, Adaptive Runtime Technology, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice.

Copyright © 1999-2006 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this publication are covered by the trademarks, service marks, or product names as designated by the companies that market those products.

Updated: 10-Apr-2006

Preface

What is Covered in this Book

This book discusses the bindings and transports supported by Artix. It describes how the combination of WSDL elements and Artix configuration is used to set-up a binding or a transport. It also discusses the advantages of using each of the bindings and transports. In the case of transports, such as Websphere MQ, it also discusses how to access some of the transports more advanced features.

Who Should Read this Book

This book is intended for people who are developing the contracts for Artix endpoints. It assumes a working knowledge of WSDL and XML. It also assumes a working knowledge of the underlying middleware technology being discussed.

How to Use this Book

This book is broken onto three parts:

- [Part I](#) provides a basic introduction to WSDL. It also provides a discussion of the WSDL elements that make up the logical portion of an Artix contract.
- [Part II](#) discusses each of the bindings supported by Artix.
- [Part III](#) discusses each of the transports supported by Artix.
- [Part IV](#) discusses using other Artix features that are driven by contract based directives. This includes the Artix router and the Artix transformer.

The Artix Library

The Artix documentation library is organized in the following sections:

- [Getting Started](#)
- [Designing and Developing Artix Solutions](#)
- [Configuring and Deploying Artix Solutions](#)
- [Using Artix Services](#)
- [Integrating Artix Solutions](#)
- [Integrating with Enterprise Management Systems](#)
- [Reference Documentation](#)

Getting Started

The books in this section provide you with a background for working with Artix. They describe many of the concepts and technologies used by Artix. They include:

- [Release Notes](#) contains release-specific information about Artix.
- [Installation Guide](#) describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- [Getting Started with Artix](#) describes basic Artix and WSDL concepts.
- [Using Artix Designer](#) describes how to use Artix Designer to build Artix solutions.
- [Artix Technical Use Cases](#) provides a number of step-by-step examples of building common Artix solutions.

Designing and Developing Artix Solutions

The books in this section go into greater depth about using Artix to solve real-world problems. They describe how Artix uses WSDL to define services, and how to use the Artix APIs to build new services. They include:

- [Building Service Oriented Architectures with Artix](#) provides an overview of service-oriented architectures and describes how they can be implemented using Artix.
- [Understanding Artix Contracts](#) describes the components of an Artix contract. Special attention is paid to the WSDL extensions used to define Artix-specific payload formats and transports.
- [Developing Artix Applications in C++](#) discusses the technical aspects of programming applications using the C++ API.
- [Developing Advanced Artix Plug-ins in C++](#) discusses the technical aspects of implementing advanced plug-ins (for example, interceptors) using the C++ API.

- [Developing Artix Applications in Java](#) discusses the technical aspects of programming applications using the Java API.

Configuring and Deploying Artix Solutions

This section includes:

- [Configuring and Deploying Artix Solutions](#) discusses how to configure and deploy Artix-enabled systems, and provides examples of typical use cases.

Using Artix Services

The books in this section describe how to use the services provided with Artix:

- [Artix Locator Guide](#) discusses how to use the Artix locator.
- [Artix Session Manager Guide](#) discusses how to use the Artix session manager.
- [Artix Transactions Guide, C++](#) explains how to enable Artix C++ applications to participate in transacted operations.
- [Artix Transactions Guide, Java](#) explains how to enable Artix Java applications to participate in transacted operations.
- [Artix Security Guide](#) explains how to use the security features of Artix.

Integrating Artix Solutions

The books in this section describe how to use Artix as a bridge between other middleware technologies and service-oriented middleware technologies.

- [Artix for CORBA](#) provides information on using Artix in a CORBA environment.
- [Artix for J2EE](#) provides information on using Artix to integrate with J2EE applications.

For details on integrating with Microsoft's .NET technology, see the documentation for Artix Connect.

Integrating with Enterprise Management Systems

The books in this section describe how to integrate Artix solutions with a range of enterprise management systems. They include:

- [IBM Tivoli Integration Guide](#) explains how to integrate Artix with IBM Tivoli.

- [BMC Patrol Integration Guide](#) explains how to integrate Artix with BMC Patrol.
- [CA WSDM Integration Guide](#) explains how to integrate Artix with CA's WSDM product.

Reference Documentation

These books provide detailed reference information about specific Artix APIs, WSDL extensions, configuration variables, command-line tools, and terminology. The reference documentation includes:

- [Artix Command Line Reference](#)
- [Artix Configuration Reference](#)
- [Artix WSDL Extension Reference](#)
- [Artix Java API Reference](#)
- [Artix C++ API Reference](#)
- [Artix .NET API Reference](#)
- [Artix Glossary](#)

Getting the Latest Version

The latest updates to the Artix documentation can be found at <http://www.iona.com/support/docs>.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Artix Library

You can search the online documentation by using the **Search** box at the top right of the documentation home page:

<http://www.iona.com/support/docs>

To search a particular library version, browse to the required index page, and use the **Search** box at the top right, for example:

<http://www.iona.com/support/docs/artix/4.0/index.xml>

You can also search within a particular book. To search within a HTML version of a book, use the **Search** box at the top left of the page. To search within a PDF version of a book, in Adobe Acrobat, select **Edit|Find**, and enter your search text.

Artix Online Help

Artix Designer and the Artix Management Console include comprehensive online help, providing:

- Step-by-step instructions on how to perform important tasks
- A full search feature
- Context-sensitive help for each screen

There are two ways that you can access the online help:

- Select **Help|Help Contents** from the menu bar. Sections on Artix Designer and the Artix Management Console appear in the contents panel of the Eclipse help browser.
- Press **F1** for context-sensitive help.

In addition, there are a number of cheat sheets that guide you through the most important functionality in Artix Designer. To access these, select **Help|Cheat Sheets**.

Artix Glossary

The [Artix Glossary](#) provides quick definitions and is a comprehensive reference for Artix terms. All terms are defined in the context of the development and deployment of Web services using Artix.

Additional Resources

The [IONA Knowledge Base](#) contains helpful articles written by IONA experts about Artix and other products.

The [IONA Update Center](#) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to [IONA Online Support](#).

Comments, corrections, and suggestions on IONA documentation can be sent to docs-support@iona.com.

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

<i>Fixed width</i>	Fixed width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the <code>IT_Bus : AnyType</code> class.
	Constant width paragraphs represent code examples or information a system displays on the screen. For example:
	<pre>#include <stdio.h></pre>
<i>Fixed width italic</i>	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:
	<pre>% cd /users/<i>YourUserName</i></pre>
<i>Italic</i>	Italic words in normal text represent <i>emphasis</i> and introduce <i>new terms</i> .
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the User Preferences dialog.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
...	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in { } (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select File Open).

PREFACE

Contents

Preface	i
What is Covered in this Book	i
Who Should Read this Book	i
How to Use this Book	i
The Artix Library	i
Getting the Latest Version	iv
Searching the Artix Library	iv
Artix Online Help	v
Artix Glossary	v
Additional Resources	v
Document Conventions	vi
List of Figures	xv
List of Tables	xvii
Part I Introduction	
Chapter 1 Introducing Artix Contracts	1
Chapter 2 Defining Logical Data Units	7
Mapping Data into Logical Data Units	9
Adding Data Units to a Contract	11
XMLSchema Simple Types	13
Defining Complex Data Types	15
Defining Data Structures	16
Defining Arrays	20
Defining Types by Extension	22
Defining Types by Restriction	23
Defining Enumerated Types	25
Defining Elements	27

Chapter 3	Defining Logical Messages Used by a Service	29
Chapter 4	Defining Your Logical Interfaces	33
 Part II Bindings		
Chapter 5	Understanding Bindings in WSDL	43
Chapter 6	Using SOAP Messages	47
	Adding a Default SOAP Binding	48
	Adding SOAP Headers to a SOAP Binding	51
	Sending Data Using SOAP with Attachments	56
Chapter 7	Using Tuxedo's FML Buffers	61
Chapter 8	Using Fixed Length Records	69
Chapter 9	Using Tagged Data	85
Chapter 10	Using Tibco Rendezvous Messages	99
	Defining a TibrvMsg Binding	101
	Artix Default Mappings for TibrvMsg	108
	Defining Array Mapping Policies	113
	Defining a Custom TibrvMsg Mapping	119
	Adding Context Information to a TibrvMsg	137
Chapter 11	Using XML Documents	141
Chapter 12	Using G2++ Messages	147

Part III Transports

Chapter 13	Understanding How Endpoints are Defined WSDL	157
Chapter 14	Using HTTP	161
	Adding an HTTP Endpoint to a Contract	162
	Configuring an HTTP Endpoint	169
	Specifying Send and Recieve Timeout Limits	170
	Specifying a Username and a Password	172
	Configuring Keep-Alive Behavior	174
	Specifying Cache Control Directives	176
Chapter 15	Using IIOP	181
Chapter 16	Using WebSphere MQ	187
	Adding a WebSphere MQ Endpoint	188
	Specifying the WebSphere Library to Load	194
	Using Queues on Remote Hosts	196
	Using WebShpere MQ's Transaction Features	198
	Setting a Value fo the Message Descriptor's Format Field	200
Chapter 17	Using the Java Messaging System	203
	Defining a JMS Endpoint	204
	Basic Endpoint Configuration	206
	Client Endpoint Configuration	210
	Server Endpoint Configuration	211
	Using the Command Line Tool	213
	Migrating to the 4.x JMS WSDL Extensions	215
	Using ActiveMQ as Your JMS Provider	216
Chapter 18	Using TIBCO Rendezvous	217
Chapter 19	Using Tuxedo	223
Chapter 20	Using FTP	227

Adding an FTP Endpoint	228
Coordinating Requests and Responses	230
Implementing the Client's Coordination Logic	231
Implementing the Server's Coordination Logic	235
Using Properties to Control Coordination Behavior	239

Part IV Other Artix Features

Chapter 21 Working with CORBA	245
Adding a CORBA Binding	246
Creating a CORBA Endpoint	252
Configuring an Artix CORBA Endpoint	253
Generating CORBA IDL	257
Chapter 22 Adding Routing Instructions	259
Artix Routing	260
Compatibility of Ports and Operations	261
Defining Routes in Artix Contracts	264
Using Port-Based Routing	265
Using Operation-Based Routing	268
Using Attribute-Based Routing	271
Using Content-Based Routing	274
Creating Routes from the Command Line	279
Advanced Routing Features	282
Load Balancing	283
Message Broadcasting	284
Failover Routing	286
Linking Routes	288
Error Handling	291
Service Lifecycles	292
Routing References to Transient Servants	294
Chapter 23 Using the Artix Transformer	297
Using the Artix Transformer as a Service	298
Using Artix to Facilitate Interface Versioning	300
WSDL Messages and the Transformer	305

Writing XSLT Scripts	308
Elements of an XSLT Script	309
XSLT Templates	311
Common XSLT Functions	317
Chapter 24 Using Codeset Conversion	319
Index	323

CONTENTS

List of Figures

Figure 1: MQ Remote Queues

197

LIST OF FIGURES

List of Tables

Table 1: complexType Descriptor Elements	17
Table 2: Part Data Type Attributes	31
Table 3: Operation Message Elements	34
Table 4: Attributes of the Input and Output Elements	35
Table 5: FML Type Support	62
Table 6: Attributes for fixed:binding	71
Table 7: Attributes for tagged:binding	86
Table 8: Attributes for tagged:operation	88
Table 9: Attributes for tagged:field	89
Table 10: Attributes for tagged:sequence	90
Table 11: Attributes for tagged:choice	92
Table 12: Attributes for tibrv:binding	102
Table 13: Attributes for tibrv:input	103
Table 14: Attributes for tibrv:output	104
Table 15: TIBCO to XSD Type Mapping	108
Table 16: Effect of tibrv:array	113
Table 17: Attributes for tibrv:array	114
Table 18: Functions Used for Specifying TibrvMsg Array Element Names	116
Table 19: Valid Casts for TibrvMsg Binding	122
Table 20: Attributes for tibrv:msg	135
Table 21: Attributes for tibrv:field	135
Table 22: Settings for CacheControl on an HTTP Server Endpoint	177
Table 23: Settings for CacheControl on HTTP Client Endpoint	179
Table 24: WebSphere MQ Server_Client Attribute Settings	194
Table 25: WebSphere MQ Transactional Attribute Settings	198
Table 26: WebSphere MQ Format Attribute Settings	200

LIST OF TABLES

Table 27: JMS Port Attributes	206
Table 28: Supported TIBCO Rendezvous Features	217
Table 29: Context QNames	271
Table 30: Context Names Used with wsdltorouting	280
Table 31: Conditions Used with wsdltorouting	280

Part I

Introduction

In this part

This part contains the following chapters:

Introducing Artix Contracts	page 1
Defining Logical Data Units	page 7
Defining Logical Messages Used by a Service	page 29
Defining Your Logical Interfaces	page 33

Introducing Artix Contracts

Artix contracts define endpoints using Web Service Description Language and a number of Artix extensions.

Overview

When using Artix to service-enable your infrastructure, you will be working directly with the WSDL and XML Schema that makes up the Artix contract. In many instances, the Artix Designer development tool automates many of the tasks involved in creating a well-formed and valid WSDL document. When hand-editing Artix contracts you will need to ensure that the contract is valid, as well as correct. To do that you must have some familiarity with WSDL. You can find the standard on the W3C web site, www.w3.org.

Structure of a WSDL document

A WSDL document is, at its simplest, a collection of elements contained within a root `definition` element. These elements describe a service and how that service can be accessed.

The `types`, `message`, and `portType` elements describe the service's interface and make up the *logical* section of a contract. Within the `types` element, XML Schema is used to define the structure of the data that makes up the messages. A number of `message` elements are used to define the structure of the messages used by the service. The `portType` element contains one or more `operation` elements that define the messages sent by the operations exposed by the service.

The `binding` and `service` elements describe how the service connects to the outside world and make up the *physical* section of the contract. `binding` elements describe how the data units defined in the `message` elements are mapped into a concrete, on-the-wire data format, such as SOAP. `service` elements contain one or more `port` elements which define the network interface for the service.

WSDL elements

A WSDL document is made up of the following elements:

- `definitions`—the root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced by the WSDL.
- `types`—the XMLSchema definitions for the data units that form the building blocks of the messages used by a service. For information about defining datatypes see [“Defining Logical Data Units” on page 7](#).
- `message`—the abstract definition of the data being communicated. These elements define the arguments of the operations making up your service. For information on defining messages see [“Defining Logical Messages Used by a Service” on page 29](#).
- `portType`—a collection of `operation` elements representing the logical interface of a service. For information about defining port types see [“Defining Your Logical Interfaces” on page 33](#).
- `operation`—the logical description of an action performed by a service. Operations are defined by the logical messages passed between two endpoints. For information on defining operations see [“Defining Your Logical Interfaces” on page 33](#).
- `binding`—the concrete data format specification for an endpoint. A binding element defines how the logical messages are mapped into the concrete data format used by an endpoint. This is where specifics such as parameter order and return values are specified. For information on defining bindings see [“Bindings” on page 41](#).
- `service`—a collection of related `port` elements. These elements are repositories for organizing endpoint definitions.

- `port`—the endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combined with the definition of transport details, and define the physical endpoint on which a service is exposed. For information on defining endpoints see [“Transports” on page 155](#).
-

Artix extensions

Artix extends the original concept of WSDL by describing services that use transports and bindings beyond SOAP over HTTP. Artix also extends WSDL to allow it to describe complex systems of services and how they are integrated. To do this IONA has extended WSDL according to the procedures outlined by W3C.

The majority of the IONA WSDL extension elements are used in the physical section of the contract because they relate to how data is mapped into an on-the-wire format and how different transports are configured. In addition, Artix defines extensions for creating routes between services, CORBA data type mappings, and working with service references.

Each extension is defined in a separate namespace and IONA provides the XML Schema definitions for each extension so that any XML editor can validate an Artix contract.

Designing a contract

To design an Artix contract for your services you must perform the following steps:

1. Define the data types used by your services.
2. Define the messages used in by your services.
3. Define the interfaces for your services.
4. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
5. Define the transport details for each of the services.
6. Define any routing rules used to connect your services.

Defining Logical Data Units

In Artix, complex data types are defined as logical units using XML Schema.

Overview

When defining a service in an Artix contract, the first thing you need to consider is how the data used as parameters for the exposed operations are going to be represented. Unlike applications that are written in a programming language that uses fixed data structures, services must define their data in logical units that can be consumed by any number of applications. This involves two steps:

1. Breaking the data into logical units that can be mapped into the data types used by the physical implementations of the service.
2. Combining the logical units into messages that are passed between endpoints to carry out the operations.

This chapter discusses the first step. [“Defining Logical Messages Used by a Service” on page 29](#) discusses the second step.

In this chapter

This chapter discusses the following topics:

Mapping Data into Logical Data Units	page 9
Adding Data Units to a Contract	page 11

XMLSchema Simple Types	page 13
Defining Complex Data Types	page 15
Defining Elements	page 27

Mapping Data into Logical Data Units

Overview

The interfaces used to implement a service define the data representing operation parameters as XML documents. If you are defining an interface for a service that is already implemented, you need to translate the data types of the implemented operations into discreet XML elements that can be assembled into a message. If you are starting from scratch, you need to determine the building blocks from which your messages are built in such a way as they make sense from an implementation standpoint.

Available type systems for defining service data units

According to the WSDL specification, you can use any type system you like to define data types in a WSDL document. However, the W3C specification states XMLSchema is the preferred canonical type system for a WSDL document. Therefore, XMLSchema is the intrinsic type system in Artix and is the only type system supported by Artix.

XMLSchema as a type system

XMLSchema is used to define how an XML document is structured. This done by defining the elements that make up the document. These elements can use native XMLSchema types, like `xsd:int`, or than can use types that are defined by the user. User defined types are either built up using combinations of XML elements or they are defined by restricting existing types. By combining type definitions and element definitions you can create intricate XML documents that can contian complex data.

When used in WSDL XMLSchema defines the structure of the XML document that will hold the data used to interact with a service. When defining the data units that your service uses, you can define them as types that specify the structure of the message parts. You can also define your data units as elements that will make up the message parts.

Considerations for creating your data units

You may consider simply creating logical data units that map directly to the types you envision using when implementing the service. While this approach will certainly work and closely follows the ideas of building and RPC-style application, it is not neccesarlily ideal for building a piece of a service-oriented architecture.

The Web Services Interoperability Organization's WS-I basic profile provides a number of guidelines for defining data units that can be seen accessed at <http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html>. In addition, the W3C also provides guidelines on using XML Schema to represent data types in WSDL documents:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.
- Define arrays using the SOAP 1.1 array encoding format.

Adding Data Units to a Contract

Overview

Depending on how you choose to create your WSDL, creating new data definitions requires varying amounts of WSDL knowledge.

Artix Designer uses wizards that generate the proper XML Schema tags for you.

If you choose to use another XML editor when writing your contract, you will need to have a much more complete understanding of XML Schema. You will also be responsible for validating your schema.

Defining types in Artix Designer

Artix Designer provides wizards to walk you through the creation of data definitions for your service. It automatically generates the elements needed for the type section of a contract and the wizards lead you through the steps to create different data definitions.

However, you will need to understand some XML Schema concepts when using Artix Designer. Also, Artix Designer does not allow you to take full advantage of XML Schema.

Using an XML editor

Defining the data used in an Artix contract involves seven steps:

1. Determine all the data units used in the interface described by the contract.
2. Create a `types` element in your contract.
3. Create a `schema` element, shown in [Example 1](#), as a child of the `type` element.

The `targetNamespace` attribute is where you specify the namespace under which your new data types are defined. The remaining entries should not be changed.

Example 1: Schema Entry for an Artix Contract

```
<schema targetNamespace="http://schemas.iona.com/bank.idl"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
```

4. For each complex type that is a collection of elements, define the data type using a `complexType` element. See [“Defining Data Structures” on page 16](#).
5. For each array, define the data type using a `complexType` element. See [“Defining Arrays” on page 20](#).
6. For each complex type that is derived from a simple type, define the data type using a `simpleType` element. See [“Defining Types by Restriction” on page 23](#).
7. For each enumerated type, define the data type using a `simpleType` element. See [“Defining Enumerated Types” on page 25](#).
8. For each element, define it using an `element` element. See [“Defining Elements” on page 27](#).

XMLSchema Simple Types

Overview

If a message part is going to be of a simple type you do not need to create a type definition for it. However, the complex types used by the interfaces defined in the contract are defined using simple types.

Entering simple types

XMLSchema simple types are mainly placed in the `element` elements used in the types section of your contract. They are also used in the `base` attribute of `restriction` elements and `extension` elements.

Simple types are always entered using the `xsd` prefix. For example, to specify that an element is of type `int`, you would enter `xsd:int` in its `type` attribute as shown in [Example 2](#).

Example 2: *Defining an Element with a Simple Type*

```
<element name="simpleInt" type="xsd:int" />
```

Supported XSD simple types

Artix supports the following XMLSchema simple types:

- `xsd:string`
- `xsd:normalizedString`
- `xsd:int`
- `xsd:unsignedInt`
- `xsd:long`
- `xsd:unsignedLong`
- `xsd:short`
- `xsd:unsignedShort`
- `xsd:float`
- `xsd:double`
- `xsd:boolean`
- `xsd:byte`
- `xsd:unsignedByte`
- `xsd:integer`
- `xsd:positiveInteger`

- `xsd:negativeInteger`
- `xsd:nonPositiveInteger`
- `xsd:nonNegativeInteger`
- `xsd:decimal`
- `xsd:dateTime`
- `xsd:time`
- `xsd:date`
- `xsd:QName`
- `xsd:base64Binary`
- `xsd:hexBinary`
- `xsd:ID`
- `xsd:token`
- `xsd:language`
- `xsd:Name`
- `xsd:NCName`
- `xsd:NMTOKEN`
- `xsd:anySimpleType`
- `xsd:anyURI`
- `xsd:gYear`
- `xsd:gMonth`
- `xsd:gDay`
- `xsd:gYearMonth`
- `xsd:gMonthDay`

Defining Complex Data Types

Overview

XMLSchema provides a flexible and powerful mechanism for building complex data structures from its simple data types. You can create data structures by creating a sequence of elements and attributes. You can also extend your defined types to create even more complex types.

In addition to allowing you to build complex data structures, you can also describe specialized types such as enumerated types, data types that have a specific range of values, or data types that need to follow certain patterns by either extending or restricting the primitive types.

In this section

This section discusses the following topics:

Defining Data Structures	page 16
Defining Arrays	page 20
Defining Types by Extension	page 22
Defining Types by Restriction	page 23
Defining Enumerated Types	page 25

Defining Data Structures

Overview

In XMLSchema, data units that are a collection of data fields are defined using `complexType` elements. The definition of a `complexType` has three parts:

1. The name of the defined type is specified in the `name` attribute of the `complexType` element.
2. The first child element of the `complexType` describes the behavior of the structure's fields when it is put on the wire. See [“complexType varieties” on page 17](#).
3. Each of the fields of the defined structure are defined in `element` elements that are grandchildren of the `complexType`. See [“Defining the parts of a structure” on page 17](#).

For example the structure shown in [Example 3](#) would be defined in XMLSchema as a `complexType` with two elements.

Example 3: Simple Structure

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

[Example 4](#) shows one possible XMLSchema mapping for `personalInfo`.

Example 4: A Complex Type

```
<complexType name="personalInfo">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </sequence>
</complexType>
```

complexType varieties

XMLSchema has three ways of describing how the fields of a complex type are organized when represented as an XML document and when passed on the wire. The first child element of the `complexType` determines which variety of complex type is being used. [Table 1](#) shows the elements used to define complex type behavior.

Table 1: *complexType Descriptor Elements*

Element	complexType Behavior
<code>sequence</code>	All the complex type's fields must be present and in the exact order they are specified in the type definition.
<code>all</code>	All the complex type's fields must be present but can be in any order.
<code>choice</code>	Only one of the elements in the structure is placed in the message.

If neither `sequence`, `all`, nor `choice` is specified, the default is `sequence`. For example, the structure defined in [Example 4](#) would generate a message containing two elements: `name` and `age`.

If the structure was defined as a `choice`, as shown in [Example 5](#), it would generate a message with either a `name` element or an `age` element.

Example 5: *Simple Complex Choice Type*

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

Defining the parts of a structure

You define the data fields that make up a structure using `element` elements. Every `complexType` should contain at least one `element`. Each `element` in the `complexType` represents a field in the defined data structure.

To fully describe a field in a data structure, `element` elements have two required attributes:

- `name` specifies the name of the data field and must be unique within the defined complex type.
- `type` specifies the type of the data stored in the field. The type can be either one of the XML Schema simple types or any named complex type that is defined in the contract.

In addition to `name` and `type`, `element` elements have two other commonly used optional attributes: `minOccurs` and `maxOccurs`. These attributes place bounds on the number of times the field occurs in the structure. By default, each field occurs only once in a complex type. Using these attributes, you can change how many times a field must or can appear in a structure. For example, you could define a field, `previousJobs`, that must occur at least three times and no more than seven times as shown in [Example 6](#).

Example 6: *Simple Complex Type with Occurrence Constraints*

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
    <element name="previousJobs" type="xsd:string"
      minOccurs="3" maxOccurs="7"/>
  </all>
</complexType>
```

You could also use `minOccurs` to make the `age` field optional by setting `minOccurs` to zero as shown in [Example 7](#). In this case `age` can be omitted and the data will still be valid.

Example 7: *Simple Complex Type with minOccurs*

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

Defining attributes

In XML documents attributes are contained in the element's tag. For example, in the `complexType` element `name` is an attribute. They are specified using the `attribute` element. It comes after the `all`, `sequence`, or `choice` element and are a direct child of the `complexType` element.

[Example 8](#) shows a complex type with an attribute.

Example 8: Complex Type with an Attribute

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string"
      minOccurs="3" maxOccurs="7"/>
  </all>
  <attribute name="age" type="xsd:int" use="optional" />
</complexType>
```

The `attribute` element has three attributes:

- `name` is a required attribute that specifies the string identifying the attribute.
- `type` specifies the type of the data stored in the field. The type can be either one of the XML Schema simple types.
- `use` specifies if the attribute is required or optional. Valid values are `required` or `optional`.

If you specify that the attribute is optional you can add the optional attribute `default`. `default` allows you to specify a default value for the attribute.

Defining Arrays

Overview

Artix supports two methods for defining arrays in a contract. The first is to define a complex type with a single element whose `maxOccurs` attribute has a value greater than one. The second is to use SOAP arrays. SOAP arrays provide added functionality such as the ability to easily define multi-dimensional arrays and transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are nothing more than a special case of a `sequence` complex type. You simply define a complex type with a single element and specify a value for the `maxOccurs` attribute. For example, to define an array of twenty floating point numbers you would use a complex type similar to the one shown in [Example 9](#).

Example 9: Complex Type Array

```
<complexType name="personalInfo">
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

You could also specify a value for `minOccurs`.

SOAP arrays

SOAP arrays are defined by deriving from the `SOAP-ENC:Array` base type using the `wsdl:arrayType`. The syntax for this is shown in [Example 10](#).

Example 10: Syntax for a SOAP Array derived using `wsdl:arrayType`

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
        wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *TypeName* specifies the name of the newly-defined array type. *ElementType* specifies the type of the elements in the array. *ArrayBounds* specifies the number of dimensions in the array. To specify a single dimension array you would use `[]`; to specify a two-dimensional array you would use either `[][]` or `[,]`.

For example, the SOAP Array, `SOAPStrings`, shown in [Example 11](#), defines a one-dimensional array of strings. The `wsdl:arrayType` attribute specifies the type of the array elements, `xsd:string`, and the number of dimensions, `[]` implying one dimension.

Example 11: *Definition of a SOAP Array*

```
<complexType name="SOAPStrings">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
        wsdl:arrayType="xsd:string []" />
    </restriction>
  </complexContent>
</complexType>
```

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in [Example 12](#).

Example 12: *Syntax for a SOAP Array derived using an Element*

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
          maxOccurs="unbounded" />
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

When using this syntax, the element's `maxOccurs` attribute must always be set to `unbounded`.

Defining Types by Extension

Overview

Like most major coding languages, XMLSchema allows you to create data types that inherit some of their elements from other data types. This is called defining a type by extension. For example, you could create a new type called `alienInfo`, that extends the `personalInfo` structure defined in [Example 4 on page 16](#) by adding a new element called `planet`.

Types defined by extension have four parts:

1. The name of the type is defined by the `name` attribute of the `complexType` element.
2. The `complexContent` element specifies that the new type will have more than one element.

Note: If you are only adding new attributes to the complex type, you can use a `simpleContent` element.

3. The type from which the new type is derived, called the *base type*, is specified in the `base` attribute of the `extension` element.
4. The new type's elements and attributes are defined in the `extension` element as they would be for a regular complex type.

For example, `alienInfo` would be defined as shown in [Example 13](#).

Example 13: Type Defined by Extension

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="personalInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Defining Types by Restriction

Overview

XMLSchema allows you to create new types by restricting the possible values of an XMLSchema simple type. For example, you could define a simple type, `SSN`, which is a string of exactly nine characters. New types defined by restricting simple types are defined using a `simpleType` element.

The definition of a `simpleType` has three parts:

1. The name of the new type is specified by the `name` attribute of the `simpleType` element.
2. The simple type from which the new type is derived, called the *base type*, is specified in the `restriction` element. See [“Specifying the base type” on page 23](#).
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the `restriction` element. See [“Defining the restrictions” on page 23](#).

Specifying the base type

The base type is the type that is being restricted to define the new type. It is specified using a `restriction` element. The `restriction` element is the only child of a `simpleType` element and has one attribute, `base`, that specifies the base type. The base type can be any of the XMLSchema simple types.

For example, to define a new type by restricting the values of an `xsd:int` you would use a definition like [Example 14](#).

Example 14: *int as Base Type*

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

Defining the restrictions

The rules defining the restrictions placed on the base type are called facets. Facets are elements with one attribute, `value`, that defines how the facet is enforced. The available facets and their valid `value` settings depend on the base type. For example, `xsd:string` supports six facets including:

- length
- minLength
- maxLength
- pattern
- whitespace
- enumeration

Each facet element is a child of the `restriction` element.

Example

Example 15 shows an example of a simple type, `SSN`, which represents a social security number. The resulting type will be a string of the form `xxx-xx-xxxx`. `<SSN>032-43-9876</SSN>` is a valid value for an element of this type, but `<SSN>032439876</SSN>` is not.

Example 15: *SSN Simple Type Description*

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

Defining Enumerated Types

Overview

Enumerated types in XMLSchema are a special case of definition by restriction. They are described by using the `enumeration` facet which is supported by all XMLSchema primitive types. As with enumerated types in most modern programming languages, a variable of this type can only have one of the specified values.

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in [Example 16](#).

Example 16: Syntax for an Enumeration

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value" />
    <enumeration value="Case2Value" />
    ...
    <enumeration value="CaseNValue" />
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

Example

For example, an XML document with an element defined by the enumeration `widgetSize`, shown in [Example 17](#), would be valid if it contained `<widgetSize>big</widgetSize>`, but not if it contained `<widgetSize>big,mungo</widgetSize>`.

Example 17: *widgetSize Enumeration*

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

Defining Elements

Overview

Elements in XMLSchema represent an instance of an element in an XML document generated from the schema. At their most basic, an element consists of a single `element` element. Like the `element` element used to define the members of a complex type, they have three attributes:

- `name` is a required attribute that specifies the name of the element as it will appear in an XML document.
- `type` specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. This attribute can be omitted if the type has an in-line definition.
- `nillable` specifies if an element can be left out of a document entirely. If `nillable` is set to `true`, the element can be omitted from any document generated using the schema.

An element can also have an *in-line* type definition. In-line types are specified using either a `complexType` element or a `simpleType` element. Once you specify if the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data. In-line type definitions are discouraged, because they are not reusable.

Defining Logical Messages Used by a Service

A service exchanges logical messages when its operations are invoked.

Overview

In an Artix contract a service's operations are defined by specifying the logical messages that are exchanged when the operation is invoked. These logical messages define the data that is passed over a network as an XML document. The logical messages are similar to the parameter lists passed into the methods of an object implementing a service.

Logical messages are defined using the `message` element in your contracts. Each logical message consists of one or more parts, defined in `part` elements. The parts of a message are similar to the individual parameters that are used by the methods of an object implementing a service.

Messages and parameter lists

Each operation exposed by a service can only have one input message and one output message. Because of this you need to carefully consider how you define the logical messages used by your service. The input message defines all of the information the service receives when the operation is invoked. The output message defines all of the data that the service sends when the operation is completed.

If you are defining an existing application as a service, you need to ensure that each parameter used by the method implementing the operation is represented by a message part. Your input message must contain one part for each input parameter in the method. Your output message must contain one part for each output parameter and a part to represent the return value if needed. If a parameter is both an input and an output parameter, it must be listed as a part of both the input message and the output message.

If you are starting from scratch, you can define your logical methods in anyway that makes sense for your application. Keep in mind, however, that when you generate code to implement your service, the method parameters generated to implement each operation will be based on the parts defined in your logical messages. For instance, if you define your input message such that it only has one part, the resulting method will only have one parameter to represent it.

Example

For example, imagine a server that stored personal information and provided a method that returned an employee's data based on an employee ID number. The method signature for looking up the data would look similar to [Example 18](#).

Example 18: *personalInfo lookup Method*

```
personalInfo lookup(long empId)
```

This method signature could be mapped to the WSDL fragment shown in [Example 19](#).

Example 19: *WSDL Message Definitions*

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
</message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo"/>
</message/>
```

Message naming

Each message in a contract must have a unique name within its namespace. It is also recommended that messages are named in a way that represents whether they are input messages that represent a service request or output messages that represent a response.

Message parts

Message parts are the formal data units of the logical message. Each part is defined using a `part` element. They are identified by a name and either a `type` attribute or an `element` attribute that specifies its data type. The data type attributes are listed in [Table 2](#)

Table 2: *Part Data Type Attributes*

Attribute	Description
<code>type="type_name"</code>	The datatype of the part is defined by a <code>simpleType</code> or <code>complexType</code> called <code>type_name</code>
<code>element="elem_name"</code>	The datatype of the part is defined by an <code>element</code> called <code>elem_name</code> .

Messages are allowed to reuse part names. For instance, if a method has a parameter, `foo`, that is passed by reference or is an in/out, it can be a part in both the request message and the response message as shown in [Example 20](#).

Example 20: Reused Part

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
</message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
</message>
```


Defining Your Logical Interfaces

Logical service interfaces are defined using the portType element.

Overview

Logical service interfaces are defined using the WSDL `portType` element. The `portType` is a collection of abstract operation definitions. Each operation is defined by the input, output, and fault messages used to complete the transaction the operation represents. When code is generated to implement the service interface defined by a `portType` element, each operation is converted into a method containing the parameters defined by the input, output, and fault messages specified in the contract.

Process

Defining a logical interface in an Artix contract entails the following:

1. Creating a `portType` element to contain the interface definition and give it a unique name. See [“Port types” on page 34](#).
2. Creating an `operation` element for each operation defined in the interface. See [“Operations” on page 34](#).
3. For each operation, specifying the messages used to represent the operation’s parameter list, return type, and exceptions. See [“Operation messages” on page 34](#).

Port types

A WSDL `portType` element is the root element in a logical interface definition. While many Web service implementations, including Artix, map `portType` elements directly to generated implementation objects, a logical interface definition does not specify the exact functionality provided by the the implemented service. For example, a logical interface named `ticketSystem` can result in an implementation that sells concert tickets or issues parking tickets.

The `portType` element is the unit of a WSDL document that is mapped into a binding to define the physical data used by an endpoint exposing the defined service. For more information on mapping logical interfaces into bindings see [“Understanding Bindings in WSDL” on page 43](#).

Each `portType` element in a WSDL document must have a unique name, specified using the `name` attribute, and is made up of a collection of operations, described in `operation` elements. A WSDL document can describe any number of port types.

Operations

Logical operations, defined using WSDL `operation` elements, define the interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation defined within a `portType` element must have a unique name, specified using the `name` attribute. The `name` attribute is required to define an operation.

Operation messages

Logical operations are made up of a set of elements representing the logical messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in [Table 3](#).

Table 3: *Operation Message Elements*

Element	Description
<code>input</code>	Specifies the message the client endpoint sends to the service provider when a request is made. The parts of this message correspond to the input parameters of the operation.

Table 3: *Operation Message Elements*

Element	Description
output	Specifies the message that the service provider sends to the client endpoint in response to a request. The parts of this message correspond to any operation parameters that can be changed by the service provider, such as values passed by reference. This includes the return value of the operation.
fault	Specifies a message used to communicate an error condition between the endpoints.

An operation is required to have at least one `input` or one `output` element. An operation can have both `input` and `output` elements, but it can only have one of each. Operations are not required to have any `fault` messages, but can have any number of `fault` messages needed.

The elements have the two attributes listed in [Table 4](#).

Table 4: *Attributes of the Input and Output Elements*

Attribute	Description
name	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
message	Specifies the abstract message that describes the data being sent or received. The value of the <code>message</code> attribute must correspond to the <code>name</code> attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the `name` attribute for all input and output elements; WSDL provides a default naming scheme based on the enclosing operation's name. If only one element is used in the operation, the element name defaults to the name of the operation. If both an `input` and an `output` element are used, the element name defaults to the name of the operation with `Request` or `Response` respectively appended to the name.

Return values

Because the `operation` element is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value it will be mapped into the

output message as the last part of that message. The concrete details of how the message parts are mapped into a physical representation are described in [“Bindings” on page 41](#).

Example

For example, you might have an interface similar to the one shown in [Example 21](#).

Example 21: *personalInfo lookup interface*

```
interface personalInfoLookup
{
    personalInfo lookup(in int empID)
    raises(idNotFound);
}
```

This interface could be mapped to the port type in [Example 22](#).

Example 22: *personalInfo lookup port type*

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

Note that the return value of `lookup()` is mapped to the message used in the `output` element of the WSDL definition. Because the operation does not have any other parameters that can be returned the return parameter is the only part of the message used for the `output` element.

Part II

Bindings

In this part

This part contains the following chapters:

Understanding Bindings in WSDL	page 43
Using SOAP Messages	page 47
Using Tuxedo's FML Buffers	page 61
Using Fixed Length Records	page 69
Using Tagged Data	page 85
Using Tibco Rendezvous Messages	page 99
Using XML Documents	page 141
Using G2++ Messages	page 147

Understanding Bindings in WSDL

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

Overview

Bindings provide a bridge between the logical messages used by a service to a concrete data format that an endpoint uses in the physical world. They describe how the logical messages are mapped into a payload format that is used on the wire by an endpoint. It is within the bindings that details such as parameter order, concrete data types, and return values are specified. For example, the parts of a message can be reordered in a binding to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

Port types and bindings

Port types and bindings are directly related. A port type is an abstract definition of a set of interactions between two logical services. A binding is a concrete definition of how the messages used to implement the logical services will be instantiated in the physical world. Each binding is then associated with a set of network details that finish the definition of one endpoint that exposes the logical service defined by the port type.

To ensure that an endpoint defines only a single service, WSDL requires that a binding can only represent a single port type. For example, if you had a contract with two port types, you could not write a single binding that mapped both of them into a concrete data format. You would need two bindings.

However, WSDL allows for a port type to be mapped to several bindings. For example, if your contract had a single port type, you could map it into two or more bindings. Each binding could alter how the parts of the message are mapped or they could specify entirely different payload formats for the message.

The WSDL elements

Bindings are defined in a contract using the WSDL `binding` element. The binding element has a single attribute, `name`, that specifies a unique name for the binding. The value of this attribute is used to associate the binding with an endpoint as discussed in [“Understanding How Endpoints are Defined WSDL” on page 157](#).

The actual mappings are defined in the children of the `binding` element. These elements vary depending on the type of payload format you decide to use. The different payload formats and the elements used to specify their mappings are discussed in the following chapters.

Adding to a contract

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

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

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different types of bindings work.

You can also add a binding to a contract using any text editor. When you hand edit a contract, you are responsible for ensuring that the contract is valid.

Supported bindings

Artix supports the following bindings:

- SOAP
- CORBA

- Fixed record length
- Pure XML
- Tagged (variable record length)
- TibrvMsg (a TIBCO Rendezvous format)
- Tuxedo's Field Manipulation Language (FML)
- G2++

Using SOAP Messages

SOAP is the standard payload format used by Web services.

Overview

Artix provides a tool to generate a default SOAP binding which does not use any SOAP headers. However, you can add SOAP headers to your binding using any text or XML editor. In addition, you can define a SOAP binding that uses MIME multipart attachments.

In this chapter

This chapter discusses the following topics:

Adding a Default SOAP Binding	page 48
Adding SOAP Headers to a SOAP Binding	page 51
Sending Data Using SOAP with Attachments	page 56

Adding a Default SOAP Binding

Overview

Artix provides three ways to add a SOAP binding for a logical interface. The first is to use the Artix Designer. The second is the command line tool `wsdltosoap` as described in [“Using wsdltosoap” on page 48](#). The third is to use the **SOAP Enable** option in the Artix Designer.

Using wsdltosoap

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

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

The command has the following options:

<code>-i portType</code>	Specifies the name of the port type being mapped to a SOAP binding.
<code>-n namespace</code>	Specifies the namespace to use for the SOAP binding.
<code>-b binding</code>	Specifies the name for the generated SOAP binding. Defaults to <code>portTypeBinding</code> .
<code>-d dir</code>	Specifies the directory into which the new WSDL file is written.
<code>-o file</code>	Specifies the name of the generated WSDL file. Defaults to <code>wsdl_file-soap.wsdl</code> .
<code>-style</code>	Specifies the encoding style to use in the SOAP binding. Defaults to <code>document</code> .
<code>-use</code>	Specifies how the data is encoded. Default is <code>literal</code> .
<code>-quiet</code>	Specifies that the tool runs in quiet mode.
<code>-verbose</code>	Specifies that the tool runs in verbose mode.
<code>-h</code>	Specifies that the tool will display a usage message.
<code>-v</code>	Displays the tool's version.

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

Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in an Artix contract similar to the one shown in [Example 23](#).

Example 23: Ordering System Interface

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

The SOAP binding generated for `orderWidgets` is shown in [Example 24](#).

Example 24: SOAP Binding for `orderWidgets`

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap:operation soapAction="" style="rpc"/>
    <input name="order">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
    </input>
    <output name="bill">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
    </output>
    <fault name="sizeFault">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
    </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the `rpc/encoded` message style. The value of the `namespace` attribute is, in this example, the same as the contract's target namespace.

Adding SOAP Headers to a SOAP Binding

Overview

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

Syntax

The syntax for defining a SOAP header is shown in [Example 25](#). The `message` attribute of `soap:header` is the qualified name of the message from which the part being inserted into the header is taken. The `part` attribute is the name of the message part inserted into the SOAP header. Because SOAP headers are always doc style, the WSDL message part inserted into the SOAP header must be defined using an element. Together the `message` and the `part` attributes fully describe the data to insert into the SOAP header.

Example 25: SOAP Header Syntax

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

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

Development considerations

When you use SOAP headers in your Artix applications, you are responsible for creating and populating the SOAP headers in your application logic. For details on Artix application development, see either [Developing Artix Applications in C++](#) or [Developing Artix Applications in Java](#).

Splitting messages between body and header

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

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

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

Example

[Example 26](#) shows a modified version of the `orderWidgets` service shown in [Example 23](#). This version has been modified so that each order has an `xsd:base64binary` value placed in the SOAP header of the request and response. The SOAP header is defined as being the `keyVal` part from the `widgetKey` message. In this case you would be responsible for adding the SOAP header in your application logic because it is not part of the input or output message.

Example 26: SOAP Binding with a SOAP Header

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

Example 26: SOAP Binding with a SOAP Header (Continued)

```

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd:keyElem"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap:operation soapAction="" style="rpc"/>
    <input name="order">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      <soap:header message="tns:widgetKey" part="keyVal"/>
    </input>
    <output name="bill">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      <soap:header message="tns:widgetKey" part="keyVal"/>
    </output>
  </operation>
</binding>

```

Example 26: SOAP Binding with a SOAP Header (Continued)

```

    <fault name="sizeFault">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
    </fault>
  </operation>
</binding>
...
</definitions>

```

You could modify [Example 26](#) so that the header value was a part of the input and output messages as shown in [Example 27](#). In this case `keyVal` is a part of the input and output messages. In the `soap:body` elements the `parts` attribute specifies that `keyVal` is not to be inserted into the body. However, it is inserted into the SOAP header.

Example 27: SOAP Binding for orderWidgets with a SOAP Header

```

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

```

Example 27: SOAP Binding for orderWidgets with a SOAP Header

```

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap:operation soapAction="" style="rpc"/>
    <input name="Order">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"
        parts="numOrdered"/>
      <soap:header message="tns:widgetOrder" part="keyVal"/>
    </input>
    <output name="bill">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"
        parts="bill"/>
      <soap:header message="tns:widgetOrderBill" part="keyVal"/>
    </output>
    <fault name="sizeFault">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
    </fault>
  </operation>
</binding>
...
</definitions>

```

Sending Data Using SOAP with Attachments

Overview

SOAP messages generally do not carry binary data. However, the W3C SOAP specification allows for using MIME multipart/related messages to send binary data in SOAP messages. This technique is called using SOAP with attachments. SOAP attachments are defined in the W3C's *SOAP Messages with Attachments Note* (<http://www.w3.org/TR/SOAP-attachments>).

Namespace

The WSDL extensions used to define the MIME multipart/related messages are defined in the namespace `http://schemas.xmlsoap.org/wsdl/mime/`. In the discussion that follows, it is assumed that this namespace is prefixed with `mime`. The entry in the WSDL `definitions` element to set this up is shown in [Example 28](#).

Example 28: MIME Namespace Specification in a Contract

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

Changing the message binding

In a default SOAP binding, the first child element of the `input`, `output`, and `fault` elements is a `soap:body` element describing the body of the SOAP message representing the data. When using SOAP with attachments, the `soap:body` element is replaced with a `mime:multipartRelated` element.

Note: WSDL does not support using `mime:multipartRelated` for fault messages.

The `mime:multipartRelated` element tells Artix that the message body is going to be a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their contents. `mime:multipartRelated` elements in Artix contain one or more `mime:part` elements that describe the individual parts of the message.

The first `mime:part` element must contain the `soap:body` element that would normally appear in a default SOAP binding. The remaining `mime:part` elements define the attachments that are being sent in the message.

Describing a MIME multipart message

MIME multipart messages are described using a `mime:multipartRelated` element that contains a number of `mime:part` elements. To fully describe a MIME multipart message in an Artix contract:

1. Inside the `input` or `output` message you want to send as a MIME multipart message, add a `mime:multipartRelated` element as the first child element of the enclosing message.
2. Add a `mime:part` child element to the `mime:multipartRelated` element and set its `name` attribute to a unique string.
3. Add a `soap:body` element as the child of the `mime:part` element and set its attributes appropriately.

If the contract had a default SOAP binding, you can copy the `soap:body` element from the corresponding message from the default binding into the MIME multipart message.

4. Add another `mime:part` child element to the `mime:multipartRelated` element and set its `name` attribute to a unique string.
5. Add a `mime:content` child element to the `mime:part` element to describe the contents of this part of the message.

To fully describe the contents of a MIME message part the `mime:content` element has the following attributes:

- ◆ `part`—Specifies the name of the WSDL message `part`, from the parent message definition, that is used as the content of this part of the MIME multipart message being placed on the wire.
- ◆ `type`—The MIME type of the data in this message part. MIME types are defined as a type and a subtype using the syntax *type/subtype*.

There are a number of predefined MIME types such as `image/jpeg` and `text/plain`. The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* (<ftp://ftp.isi.edu/in-notes/rfc2045.txt>) and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* (<ftp://ftp.isi.edu/in-notes/rfc2046.txt>).

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

Example

[Example 29](#) shows an Artix contract for a service that stores X-rays in JPEG format. The image data, `xRay`, is stored as an `xsd:base64binary` and is packed into the MIME multipart message's second part, `imageData`. The remaining two parts of the input message, `patientName` and `patientNumber`, are sent in the first part of the MIME multipart image as part of the SOAP body.

Example 29: Contract using SOAP with Attachments

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
  targetNamespace="http://mediStor.org/x-rays"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://mediStor.org/x-rays"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<message name="storRequest">
  <part name="patientName" type="xsd:string"/>
  <part name="patientNumber" type="xsd:int"/>
  <part name="xRay" type="xsd:base64Binary"/>
</message>
<message name="storResponse">
  <part name="success" type="xsd:boolean"/>
</message>
<portType name="xRayStorage">
  <operation name="store">
    <input message="tns:storRequest" name="storRequest"/>
    <output message="tns:storResponse" name="storResponse"/>
  </operation>
</portType>
```

Example 29: *Contract using SOAP with Attachments*

```

<binding name="xRayStorageBinding" type="tns:xRayStorage">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="store">
    <soap:operation soapAction="" style="rpc"/>
    <input name="storRequest">
      <mime:multipartRelated>
        <mime:part name="bodyPart">
          <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://mediStor.org/x-rays" use="encoded"/>
        </mime:part>
        <mime:part name="imageData">
          <mime:content part="xRay" type="image/jpeg"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
    <output name="storResponse">
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:AttachmentService" use="encoded"/>
    </output>
  </operation>
</binding>
<service name="xRayStorageService">
  <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
    <soap:address location="http://localhost:9000"/>
  </port>
</service>
</definitions>

```


Using Tuxedo's FML Buffers

Artix can send and receive messages packaged as FML buffers.

Overview

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

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

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

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

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

FML/XML Schema support

An FML buffer can only contain the data types listed in [Table 5](#).

Table 5: *FML Type Support*

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

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

Mapping from a field table to an Artix contract

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

1. Create the logical data representation of the FML buffer in the Artix contract as described in [“Mapping to logical type descriptions” on page 62](#).
2. Enter the FML binding information using Artix WSDL extensors as described in [“Adding the FML binding” on page 67](#).

Mapping to logical type descriptions

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

1. If the C header file for the FML buffer does not exist, generate it from the field table using the Tuxedo `mkfldhdr` or `mkfldhdr32` utility program.
2. For each field in the FML buffer, create an `element` with the following attribute settings:

- ◆ `name` is set to the name specified in the field table.
 - ◆ `type` is set to the appropriate XML Schema type for the type specified in the field table. See [“FML/XML Schema support” on page 62.](#)
3. If your Tuxedo application has data fields that are always used together, you can group the corresponding elements into complex types.

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

For example, consider a Tuxedo application that returns personnel records on employees that needs to be exposed through a new web interface. The Tuxedo application uses the field table file shown in [Example 30.](#)

Example 30: *personnelInfo Field Table File*

```
# personnelInfo Field Table
# name      number  type      flags      comment
empId       100      long      -          -
name        101      string    -          -
age         102      short     -          -
dept        103      string    -          -
addr        104      string    -          -
city        105      string    -          -
state       106      string    -          -
zip         107      string    -          -
```

The C++ header file generated by the Tuxedo `mkfldhdr` tool to represent the `personnelInfo` FML buffer is shown in [Example 31](#). Even if you are not planning to access the FML buffer using the compile-time method, you will need to generate the header file when using Artix because this will give you the `fldid` values for the fields in the buffer.

Example 31: *personnelInfo C++ header*

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

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

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

Because the employee data is always returned as a unit you can group it into a complex type as shown in [Example 32](#).

Example 32: *Logical description of personnelInfo FML buffer*

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

The interface for your Tuxedo application would be mapped to a `portType` similar to [Example 33](#).

Example 33: *personnelInfo Lookup Interface*

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

Example 33: *personnelInfo Lookup Interface*

```

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

```

Flattened XML and FML

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

As a result, complex types defined in XML Schema are collapsed into their composite elements. For instance, the message `lookupResponse`, which uses the complex type defined in [Example 32 on page 65](#), would be equivalent to the message definition in [Example 34](#) when processed by the FML binding.

Example 34: *Flattened Message for FML*

```

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

```

Adding the FML binding

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

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

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

2. Create a new `binding` element in your contract to define the FML buffer's binding.
3. Add a `tuxedo:binding` element to identify that this binding defines an FML buffer.
4. Add a `tuxedo:fieldTable` element to the binding to describe how the element names defined in the logical portion of the contract map to the `fldid` values for the corresponding fields in the FML buffer.

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

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

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

- ◆ `name` specifies the name of the logical type describing the field.
- ◆ `id` specifies the `fldid` value for the field in the FML buffer.

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

For example, the binding for the `personalInfo` FML buffer, defined in [Example 30 on page 63](#), will be similar to the binding shown in [Example 35](#).

Example 35: *personalInfo FML binding*

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

Using Fixed Length Records

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

Overview

The Artix fixed binding is used to represent fixed record length data. Common uses for this type of payload format are communicating with back-end services on mainframes and applications written in COBOL. Artix provides several means for creating a contract containing a fixed binding:

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

Hand editing

To map a logical interface to a fixed binding:

1. Add the proper namespace reference to the `definition` element of your contract. See [“Fixed binding namespace” on page 70](#).
2. Add a WSDL `binding` element to your contract to hold the fixed binding, give the binding a unique `name`, and specify the port type that represents the interface being bound.
3. Add a `fixed:binding` element as a child of the new `binding` element to identify this as a fixed binding and set the element’s attributes to properly configure the binding. See [“fixed:binding” on page 71](#).
4. For each operation defined in the bound interface, add a WSDL `operation` element to hold the binding information for the operation’s messages.
5. For each operation added to the binding, add a `fixed:operation` child element to the `operation` element. See [“fixed:operation” on page 71](#).
6. For each operation added to the binding, add the `input`, `output`, and `fault` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the port type definition of the logical operation.
7. For each `input`, `output`, and `fault` element in the binding, add a `fixed:body` child element to define how the message parts are mapped into the concrete fixed record length payload. See [“fixed:body” on page 71](#).

Fixed binding namespace

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

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

fixed:binding

`fixed:binding` specifies that the binding is for fixed record length data. Its attributes are described in [Table 6](#).

Table 6: *Attributes for fixed:binding*

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

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

fixed:operation

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

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

fixed:body

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

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

1. If the default justification, padding, or encoding settings for the attribute are not correct for this particular message, override them by setting the following optional attributes for `fixed:body`.
 - ◆ `justification` specifies how the data in the messages are justified. Valid values are `left` and `right`.
 - ◆ `encoding` specifies the codeset used to encode text data. Valid values are any valid ISO locale or IANA codeset name.
 - ◆ `padHexCode` specifies the hex value of the character used to pad the record.
2. For each part in the message the `fixed:body` element is binding, add the appropriate child element to define the part's concrete format on the wire.

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

- ◆ [fixed:field](#) maps message parts defined using a simple type. See “XMLSchema Simple Types” on page 13.
 - ◆ [fixed:sequence](#) maps message parts defined using a sequence complex type. Complex types defined using `all` are not supported by the fixed format binding. See “Defining Data Structures” on page 16.
 - ◆ [fixed:choice](#) maps message parts defined using a choice complex type. See “Defining Data Structures” on page 16.
3. If you need to add any fields that are specific to the binding and that will not be passed to the applications, define them using a [fixed:field](#) element with its `bindingOnly` attribute set to `true`.

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

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

fixed:field

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

1. Create a `fixed:field` child element to the `fixed:body` element representing the message.
2. Set the `fixed:field` element's `name` attribute to the name of the message part defined in the logical message description that this element is mapping.
3. If the data being mapped is of type `xsd:string`, a simple type that has `xsd:string` as its base type, or an enumerated type set, the `size` attribute of the `fixed:field` element.

Note: If the message part is going to hold a date you can opt to use the `format` attribute described in step 4 instead of the `size` attribute.

`size` specifies the length of the string record in the concrete fixed message. For example, the logical message part, `raverID`, described in [Example 36](#) would be mapped to a `fixed:field` similar to [Example 37](#).

Example 36: Fixed String Message

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

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

Example 37: Fixed String Mapping

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

4. If the data being mapped is of a numerical type, like `xsd:int`, or a simple type that has a numerical type as its base type, set the `fixed:field` element's `format` attribute.
`format` specifies how non-string data is formatted. For example, if a field contains a 2-digit numeric value with one decimal place, it would

be described in the logical part of the contract as an `xsd:float`, as shown in [Example 38](#).

Example 38: *Fixed Record Numeric Message*

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

From the logical description of the message, Artix has no way of determining that the value of `rageLevel` is a 2-digit number with one decimal place because the fixed record length binding treats all data as characters. When mapping `rageLevel` in the fixed binding you would specify its `format` with `##.##`, as shown in [Example 39](#). This provides Artix with the meta-data needed to properly handle the data.

Example 39: *Mapping Numerical Data to a Fixed Binding*

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

Dates are specified in a similar fashion. For example, the `format` of the date 12/02/72 is `MM/DD/YY`. When using the fixed binding it is recommended that dates are described in the logical part of the contract using `xsd:string`. For example, a message containing a date would be described in the logical part of the contract as shown in [Example 40](#).

Example 40: *Fixed Date Message*

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

If `goDate` is entered using the standard short date format for US English locales, `mm/dd/yyyy`, you would map it to a fixed record field as shown in [Example 41](#).

Example 41: *Fixed Format Date Mapping*

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

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

`fixedValue` specifies a static value to be passed on the wire. When used without `bindingOnly="true"`, the value specified by `fixedValue` replaces any data that is stored in the message part passed to the fixed record binding. For example, if `goDate`, shown in [Example 40 on page 74](#), were mapped to the fixed field shown in [Example 42](#), the actual message returned from the binding would always have the date 11/11/2112.

Example 42: *fixedValue Mapping*

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

6. If the data being mapped is of an enumerated type, see [“Defining Enumerated Types” on page 25](#), add a `fixed:enumeration` child element to the `fixed:field` element for each possible value of the enumerated type.

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

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

Example 43: *Ice Cream Enumeration*

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

When you map the enumerated type, you need to know the concrete representation for each of the enumerated values. The concrete representations can be identical to the logical or some other value. The enumerated type in [Example 43](#) could be mapped to the fixed field shown in [Example 44](#). Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to `OrangeTango`.

Example 44: *Fixed Ice Cream Mapping*

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

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

fixed:choice

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

1. Add a `fixed:choice` child element to the `fixed:body` element.
2. Set the `fixed:choice` element's `name` attribute to the name of the logical message part being mapped.
3. Set the `fixed:choice` element's optional `discriminatorName` attribute to the name of the field used as the discriminator for the union.

The value for `discriminatorName` corresponds to the name of a `bindingOnly fixed:field` element that describes the type used for the union's discriminator as shown in [Example 45](#). The only restriction in describing the discriminator is that it must be able to handle the values

used to determine the case of the union. Therefore the values used in the union mapped in [Example 45](#) must be two-digit integers.

Example 45: Using discriminatorName

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

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

fixed:case

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

1. Set the `fixed:case` element's `name` attribute to the name of the logical definition's element.
2. Set the `fixed:case` element's `fixedValue` attribute to the value of the discriminator that selects this element. The value of `fixedValue` must correspond to the format specified by the `discriminatorName` attribute of the parent `fixed:choice` element.
3. Add a child element to define how the element's data is mapped into a fixed record.

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

Example 46 shows an Artix contract fragment mapping a choice complex type to a fixed record length message.

Example 46: *Mapping a Union to a Fixed Record Length Message*

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

Example 46: Mapping a Union to a Fixed Record Length Message

```
<fixed:choice name="stationPart"
  discriminatorName="disc">
  <fixed:case name="train" fixedValue="01">
    <fixed:field name="name" size="20"/>
  </fixed:case>
  <fixed:case name="bus" fixedValue="02">
    <fixed:field name="number" format="###"/>
  </fixed:case>
  <fixed:case name="cab" fixedValue="03">
    <fixed:field name="number" format="###"/>
  </fixed:case>
  <fixed:case name="subway" fixedValue="04">
    <fixed:field name="name" format="10"/>
  </fixed:case>
</fixed:choice>
...
</binding>
...
</definition>
```

fixed:sequence

`fixed:sequence` maps sequence complex types to a fixed record length message. To map a sequence complex type to a `fixed:sequence`:

1. Add a `fixed:sequence` child element to the `fixed:body` element.
2. Set the `fixed:sequence` element's `name` attribute to the name of the logical message part being mapped.
3. For each element in the logical definition of the message part, add a child element to define the mapping for the part's type to the physical fixed message.

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

4. If any elements of the logical data definition have occurrence constraints, see ["Defining Data Structures" on page 16](#), map the element into a `fixed:sequence` element with its `occurs` and `counterName` attributes set.

The `occurs` attribute specifies the number of times this sequence occurs in the message buffer. `counterName` specifies the name of the field used for specifying the number of sequence elements that are actually being sent in the message. The value of `counterName` corresponds to a binding-only `fixed:field` with at least enough digits to count to the value specified in `occurs` as shown in [Example 47](#). The value passed to the counter field can be any number up to the value specified by `occurs` and allows operations to use less than the specified number of sequence elements. Artix will pad out the sequence to the number of elements specified by `occurs` when the data is transmitted to the receiver so that the receiver will get the data in the promised fixed format.

Example 47: *Using counterName*

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

For example, a structure containing a name, a date, and an ID number would contain three `fixed:field` elements to fully describe the mapping of the data to the fixed record message. [Example 48](#) shows an Artix contract fragment for such a mapping.

Example 48: *Mapping a Sequence to a Fixed Record Length Message*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
  targetNamespace="http://www.iona.com/FixedService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:fixed="http://schemas.iona.com/bindings/fixed"
  xmlns:tns="http://www.iona.com/FixedService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/FixedService"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Example 48: Mapping a Sequence to a Fixed Record Length Message

```
<xsd:complexType name="person">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="date" type="xsd:string"/>
    <xsd:element name="ID" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>
...
</types>
<message name="fixedSequence">
  <part name="personPart" type="tns:person"/>
</message>
<portType name="fixedSequencePortType">
  ...
</portType>
<binding name="fixedSequenceBinding"
  type="tns:fixedSequencePortType">
  <fixed:binding/>
  ...
  <fixed:sequence name="personPart">
    <fixed:field name="name" size="20"/>
    <fixed:field name="date" format="MM/DD/YY"/>
    <fixed:field name="ID" format="#####"/>
  </fixed:sequence>
  ...
</binding>
...
</definition>
```

Example

Example 49 shows an example of an Artix contract containing a fixed record length message binding.

Example 49: Fixed Record Length Message Binding

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

Example 49: *Fixed Record Length Message Binding*

```

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
    <xsd:simpleType name="widgetSize">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="big"/>
        <xsd:enumeration value="large"/>
        <xsd:enumeration value="mungo"/>
        <xsd:enumeration value="gargantuan"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Address">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street1" type="xsd:string"/>
        <xsd:element name="street2" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zipCode" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderBillInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="amtDue" type="xsd:float"/>
        <xsd:element name="orderNumber" type="xsd:string"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
  </schema>
</types>
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>

```

Example 49: Fixed Record Length Message Binding

```
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <fixed:binding/>
  <operation name="placeWidgetOrder">
    <fixed:operation discriminator="widgetDisc"/>
    <input name="widgetOrder">
      <fixed:body>
        <fixed:sequence name="widgetOrderForm">
          <fixed:field name="amount" format="###"/>
          <fixed:field name="order_date" format="MM/DD/YYYY"/>
          <fixed:field name="type" size="2">
            <fixed:enumeration value="big" fixedValue="bg"/>
            <fixed:enumeration value="large" fixedValue="lg"/>
            <fixed:enumeration value="mungo" fixedValue="mg"/>
            <fixed:enumeration value="gargantuan" fixedValue="gg"/>
          </fixed:field>
          <fixed:sequence name="shippingAddress">
            <fixed:field name="name" size="30"/>
            <fixed:field name="street1" size="100"/>
            <fixed:field name="street2" size="100"/>
            <fixed:field name="city" size="20"/>
            <fixed:field name="state" size="2"/>
            <fixed:field name="zip" size="5"/>
          </fixed:sequence>
        </fixed:sequence>
      </fixed:body>
    </input>
```

Example 49: *Fixed Record Length Message Binding*

```

<output name="widgetOrderBill">
  <fixed:body>
    <fixed:sequence name="widgetOrderConformation">
      <fixed:field name="amount" format="###"/>
      <fixed:field name="order_date" format="MM/DD/YYYY"/>
      <fixed:field name="type" size="2">
        <fixed:enumeration value="big" fixedValue="bg"/>
        <fixed:enumeration value="large" fixedValue="lg"/>
        <fixed:enumeration value="mungo" fixedValue="mg"/>
        <fixed:enumeration value="gargantuan" fixedValue="gg"/>
      </fixed:field>
      <fixed:field name="amtDue" format="####.##"/>
      <fixed:field name="orderNumber" size="20"/>
      <fixed:sequence name="shippingAddress">
        <fixed:field name="name" size="30"/>
        <fixed:field name="street1" size="100"/>
        <fixed:field name="street2" size="100"/>
        <fixed:field name="city" size="20"/>
        <fixed:field name="state" size="2"/>
        <fixed:field name="zip" size="5"/>
      </fixed:sequence>
    </fixed:sequence>
  </fixed:body>
</output>
</operation>
</binding>
<service name="orderWidgetsService">
  <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
    <http:address location="http://localhost:8080"/>
  </port>
</service>
</definitions>

```

Using Tagged Data

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

Overview

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

Artix provides several ways of creating a contract with a tagged binding:

- The Artix Designer can create a contract with a tagged binding from a description of the tagged data.
- The Artix Designer can create a tagged binding for an existing interface.
- You can enter the binding information using any text editor or XML editor.

Hand editing

To map a logical interface to a tagged data format:

1. Add the proper namespace reference to the `definition` element of your contract. See [“Tagged binding namespace” on page 86](#).
2. Add a WSDL `binding` element to your contract to hold the tagged binding, give the binding a unique name, and specify the port type that represents the interface being bound.

3. Add a `tagged:binding` element as a child of the new `binding` element to identify this as a tagged binding and set the element's attributes to properly configure the binding.
4. For each operation defined in the bound interface, add a WSDL `operation` element to hold the binding information for the operation's messages.
5. For each operation added to the binding, add a `tagged:operation` child element to the `operation` element.
6. For each operation added to the binding, add the `input`, `output`, and `fault` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the port type definition of the logical operation.
7. For each `input`, `output`, and `fault` element in the binding, add a `tagged:body` child element to define how the message parts are mapped into the concrete tagged data payload.

Tagged binding namespace

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

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

`tagged:binding`

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

Table 7: *Attributes for `tagged:binding`*

Attribute	Purpose
<code>selfDescribing</code>	Required attribute specifying if the message data on the wire includes the field names. Valid values are <code>true</code> or <code>false</code> . If this attribute is set to <code>false</code> , the setting for <code>fieldNameValueSeparator</code> is ignored.

Table 7: *Attributes for tagged:binding*

Attribute	Purpose
fieldSeparator	Required attribute that specifies the delimiter the message uses to separate fields. Supported values are <code>newline(\n)</code> , <code>comma(,)</code> , <code>semicolon(,)</code> , and <code>pipe()</code> .
fieldNameValueSeparator	Specifies the delimiter used to separate field names from field values in self-describing messages. Supported values are: <code>equals(=)</code> , <code>tab(\t)</code> , and <code>colon(:)</code> .
scopeType	Specifies the scope identifier for complex messages. Supported values are <code>tab(\t)</code> , <code>curlybrace({data})</code> , and <code>none</code> . The default is <code>tab</code> .
flattened	Specifies if data structures are flattened when they are put on the wire. If <code>selfDescribing</code> is <code>false</code> , then this attribute is automatically set to <code>true</code> .
messageStart	Specifies a special token at the start of a message. It is used when messages require a special character at the start of a the data sequence. Currently the only supported value is <code>star(*)</code> .
messageEnd	Specifies a special token at the end of a message. Supported values are <code>newline(\n)</code> and <code>percent(%)</code> .
unscopedArrayElement	Specifies if array elements need to be scoped as children of the array. If set to <code>true</code> , arrays take the form <code>echoArray{myArray=2;item=abc;item=def}</code> . If set to <code>false</code> , arrays take the form <code>echoArray{myArray=2;{0=abc;1=def;}}</code> . Default is <code>false</code> .
ignoreUnknownElements	Specifies if Artix ignores undefined elements in the message payload. Default is <code>false</code> .

Table 7: *Attributes for tagged:binding*

Attribute	Purpose
<code>ignoreCase</code>	Specifies if Artix ignores the case with element names in the message payload. Default is <code>false</code> .

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

tagged:operation

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

Table 8: *Attributes for tagged:operation*

Attribute	Purpose
<code>discriminator</code>	Specifies a discriminator for identifying the operation as it is sent down the wire by the Artix runtime.
<code>discriminatorStyle</code>	Specifies how the discriminator will identify data as it is sent down the wire by the Artix runtime. Supported values are <code>msgname</code> , <code>partlist</code> , and <code>fieldname</code> .

tagged:body

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

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

- [tagged:field](#)
- [tagged:sequence](#)
- [tagged:choice](#)

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

tagged:field

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

Table 9: *Attributes for tagged:field*

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

When describing enumerated types `tagged:field` will have a number of [tagged:enumeration](#) child elements.

tagged:enumeration

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

For example, if you had an enumerated type, `flavorType`, with the values `FruityTooty`, `Rainbow`, `BerryBomb`, and `OrangeTango` the logical description of the type would be similar to [Example 50](#).

Example 50: Ice Cream Enumeration

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

flavorType would be mapped to the tagged data format shown in [Example 51](#).

Example 51: *Tagged Data Ice Cream Mapping*

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

tagged:sequence

tagged:sequence maps arrays and sequences to a tagged data format. Its three attributes are described in [Table 10](#).

Table 10: *Attributes for tagged:sequence*

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

A tagged:sequence can contain any number of [tagged:field](#), [tagged:sequence](#), or [tagged:choice](#) child elements to describe the data contained within the sequence being mapped. For example, a structure containing a name, a date, and an ID number would contain three

tagged:field elements to fully describe the mapping of the data to the fixed record message. [Example 52](#) shows an Artix contract fragment for such a mapping.

Example 52: *Mapping a Sequence to a Tagged Data Format*

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

tagged:choice

`tagged:choice` maps unions to a tagged data format. Its three attributes are described in [Table 11](#).

Table 11: *Attributes for tagged:choice*

Attributes	Purpose
<code>name</code>	A required attribute that must correspond to the name of the logical message part that is being mapped to the tagged data union.
<code>discriminatorName</code>	Specifies the message part used as the discriminator for the union.
<code>alias</code>	An optional attribute specifying an alias for the union that can be used to identify it on the wire.

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

tagged:case

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

`tagged:case` must contain one child element to describe the mapping of the case's data to a tagged data format. Valid child elements are `tagged:field`, `tagged:sequence`, and `tagged:choice`. [Example 53](#) shows an Artix contract fragment mapping a union to a tagged data format.

Example 53: *Mapping a Union to a Tagged Data Format*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
  targetNamespace="http://www.iona.com/tagService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:fixed="http://schemas.iona.com/bindings/tagged"
  xmlns:tns="http://www.iona.com/tagService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Example 53: Mapping a Union to a Tagged Data Format

```
<types>
  <schema targetNamespace="http://www.iona.com/tagService"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
    <xsd:complexType name="unionStationType">
      <xsd:choice>
        <xsd:element name="train" type="xsd:string"/>
        <xsd:element name="bus" type="xsd:int"/>
        <xsd:element name="cab" type="xsd:int"/>
        <xsd:element name="subway" type="xsd:string"/>
      </xsd:choice>
    </xsd:complexType>
    ...
  </types>
  <message name="tagUnion">
    <part name="stationPart" type="tns:unionStationType"/>
  </message>
  <portType name="tagUnionPortType">
    ...
  </portType>
  <binding name="tagUnionBinding" type="tns:tagUnionPortType">
    <tagged:binding selfDescribing="false"
      fieldSeparator="comma"/>
    ...
    <tagged:choice name="stationPart" discriminatorName="disc">
      <tagged:case name="train">
        <tagged:field name="name"/>
      </tagged:case>
      <tagged:case name="bus">
        <tagged:field name="number"/>
      </tagged:case>
      <tagged:case name="cab">
        <tagged:field name="number"/>
      </tagged:case>
      <tagged:case name="subway">
        <tagged:field name="name"/>
      </tagged:case>
    </tagged:choice>
    ...
  </binding>
  ...
</definition>
```

Example

Example 54 shows an example of an Artix contract containing a tagged data format binding.

Example 54: *Tagged Data Format Binding*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:taged="http://schemas.iona.com/binings/tagged"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:simpleType name="widgetSize">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="big"/>
          <xsd:enumeration value="large"/>
          <xsd:enumeration value="mungo"/>
          <xsd:enumeration value="gargantuan"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:complexType name="Address">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street1" type="xsd:string"/>
          <xsd:element name="street2" type="xsd:string"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zipCode" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="widgetOrderInfo">
        <xsd:sequence>
          <xsd:element name="amount" type="xsd:int"/>
          <xsd:element name="order_date" type="xsd:string"/>
          <xsd:element name="type" type="xsd1:widgetSize"/>
          <xsd:element name="shippingAddress" type="xsd1:Address"/>
        </xsd:sequence>
      </xsd:complexType>
    </schema>
  </types>
</definitions>
```

Example 54: Tagged Data Format Binding

```
<xsd:complexType name="widgetOrderBillInfo">
  <xsd:sequence>
    <xsd:element name="amount" type="xsd:int"/>
    <xsd:element name="order_date" type="xsd:string"/>
    <xsd:element name="type" type="xsd1:widgetSize"/>
    <xsd:element name="amtDue" type="xsd:float"/>
    <xsd:element name="orderNumber" type="xsd:string"/>
    <xsd:element name="shippingAddress" type="xsd1:Address"/>
  </xsd:sequence>
</xsd:complexType>
</schema>
</types>
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
```

Example 54: *Tagged Data Format Binding*

```

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <tagged:binding selfDescribing="false" fieldSeparator="pipe"/>
  <operation name="placeWidgetOrder">
    <tagged:operation discriminator="widgetDisc"/>
    <input name="widgetOrder">
      <tagged:body>
        <tagged:sequence name="widgetOrderForm">
          <tagged:field name="amount"/>
          <tagged:field name="order_date"/>
          <tagged:field name="type" >
            <tagged:enumeration value="big"/>
            <tagged:enumeration value="large"/>
            <tagged:enumeration value="mungo"/>
            <tagged:enumeration value="gargantuan"/>
          </tagged:field>
          <tagged:sequence name="shippingAddress">
            <tagged:field name="name"/>
            <tagged:field name="street1"/>
            <tagged:field name="street2"/>
            <tagged:field name="city"/>
            <tagged:field name="state"/>
            <tagged:field name="zip"/>
          </tagged:sequence>
        </tagged:sequence>
      </tagged:body>
    </input>
  </operation>
</binding>

```

Example 54: Tagged Data Format Binding

```
<output name="widgetOrderBill">
  <tagged:body>
    <tagged:sequence name="widgetOrderConformation">
      <tagged:field name="amount"/>
      <tagged:field name="order_date"/>
      <tagged:field name="type">
        <tagged:enumeration value="big"/>
        <tagged:enumeration value="large"/>
        <tagged:enumeration value="mungo"/>
        <tagged:enumeration value="gargantuan"/>
      </tagged:field>
      <tagged:field name="amtDue"/>
      <tagged:field name="orderNumber"/>
      <tagged:sequence name="shippingAddress">
        <tagged:field name="name"/>
        <tagged:field name="street1"/>
        <tagged:field name="street2"/>
        <tagged:field name="city"/>
        <tagged:field name="state"/>
        <tagged:field name="zip"/>
      </tagged:sequence>
    </tagged:sequence>
  </tagged:body>
</output>
</operation>
</binding>
<service name="orderWidgetsService">
  <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
    <http:address location="http://localhost:8080"/>
  </port>
</service>
</definitions>
```


Using Tibco Rendezvous Messages

Artix can natively use the Tibco TibrvMsg data format to send and receive messages.

Overview

Tibco Rendezvous applications typically use a Tibco specific data format called a TibrvMsg. Artix provides a very flexible mechanism for mapping messages into the TibrvMsg format. This allows you to integrate with existing Tibco/RV applications by service-enabling them.

The TibrvMsg binding provides default mappings for most XML Schema constructs to simplify defining a TibrvMsg in an Artix contract. The TibrvMsg binding also supports custom mappings between the messages defined in an Artix contract and the physical representation of a TibrvMsg. Custom mappings also support the inclusion of static binding-only data.

To further extend the functionality of the TibrvMsg binding, Artix includes a mechanism for passing context data stored in an Artix application as part of a TibrvMsg. For more information about using Artix contexts see either [Developing Artix Applications in C++](#) or [Developing Artix Applications in Java](#).

In this chapter

This chapter discusses the following topics:

Defining a TibrvMsg Binding	page 101
Artix Default Mappings for TibrvMsg	page 108
Defining Array Mapping Policies	page 113
Defining a Custom TibrvMsg Mapping	page 119
Adding Context Information to a TibrvMsg	page 137

Defining a TibrvMsg Binding

Overview

The Artix TibrvMsg binding provides a set of default mappings to make writing a binding simple. By default, messages are mapped into a root TibrvMsg such that parts defined using XML Schema native types become TibrvMsgFields of the root TibrvMsg and parts defined using complex types become TibrvMsgs within the root message. The elements comprising a complex type also follow the same default mapping behavior. The default mappings will work for most basic applications. For a detailed explanation of how WSDL types are mapped to TibrvMsg see [“Artix Default Mappings for TibrvMsg” on page 108](#).

Procedure

To map a logical interface to a TibrvMsg:

1. Add the proper namespace reference to the `definition` element of your contract. See [“TibrvMsg binding namespace” on page 102](#).
2. Add a WSDL `binding` element to your contract to hold the TibrvMsg binding, give the binding a unique `name`, and specify the port type that represents the interface being bound.
3. Add a `tibrv:binding` element as a child of the new `binding` element to identify this as a TibrvMsg binding and specify any global parameters.
4. For each operation defined in the bound interface, add a WSDL `operation` element to hold the binding information for the operation’s messages.
5. For each operation in the binding, add a `tibrv:operation` child element and set its attributes.
6. For each operation in the binding, add the `input`, `output`, and `fault` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the port type definition of the logical operation.
7. For each `input` element in the binding, add a `tibrv:input` child element and set its attributes.
8. For each `output` element in the binding, add a `tibrv:output` child element and set its attributes.

9. To add custom message mappings see [“Defining a Custom TibrvMsg Mapping” on page 119](#).

TibrvMsg binding namespace

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

```
xmlns:tibrv="http://schemas.iona.com/transport/tibrv"
```

tibrv:binding

`tibrv:binding` is an immediate child of the WSDL `binding` element and identifies that the data is to be packed into a TibrvMsg. Its attributes are described in [Table 12](#).

Table 12: *Attributes for tibrv:binding*

Attribute	Purpose
<code>stringEncoding</code>	An optional attribute that specifies the character set used in encoding string data included in the message. The default value is <code>utf-8</code> .
<code>stringAsOpaque</code>	An optional attribute that specifies how string data is passed in messages. <code>false</code> , the default value, specifies that string data is passed as <code>TIBRVMSG_STRING</code> . <code>true</code> specifies that string data is passed as <code>OPAQUE</code> .

In addition to the above properties, `tibrv:binding` can also specify a policy for how array data is handled for messages using the binding. The array policy is set using a child `tibrv:array` element. The array policy set at the binding level can be overridden on a per-operation basis, per-message basis, and a per-type basis. For information on defining array policies see [“Defining Array Mapping Policies” on page 113](#).

The `tibrv:binding` element can also define binding-only message data using the `tibrv:msg` element, the `tibrv:field` element, or `tibrv:context` element. Any binding-only data defined at the binding level is attached to all messages that use the binding.

tibrv:operation

`tibrv:operation` is the immediate child of a WSDL `operation` element. `tibrv:operation` has no attributes. It can, however, specify an operation-specific array policy using a child `tibrv:array` element. This array policy overrides any array policy set at the binding level. For information on defining array policies see [“Defining Array Mapping Policies” on page 113](#).

Within a `tibrv:operation` element you can also define binding-only message data using the `tibrv:msg` element, the `tibrv:field` element, or `tibrv:context` element. Any binding-only data defined at the operation level is attached to all messages that make up the operation.

tibrv:input

`tibrv:input` is the immediate child of a WSDL `input` element and defines a number of properties used in mapping the input message to a TibrvMsg. Its attributes are described in [Table 13](#).

Table 13: *Attributes for tibrv:input*

Attribute	Purpose
<code>messageNameFieldPath</code>	An optional attribute that specifies the field path that includes the message name. If this attribute is not specified, the first field in the top level message will be used as the message name and given the value <code>IT_BUS_MESSAGE_NAME</code> .
<code>messageNameFieldValue</code>	An optional attribute that specifies the field value that corresponds to the message name. If this attribute is not specified, the WSDL message’s name will be used.
<code>stringEncoding</code>	An optional attribute that specifies the character set used in encoding string data included in the message. This value will override the value set in tibrv:binding .

Table 13: *Attributes for `tibrv:input`*

Attribute	Purpose
<code>stringAsOpaque</code>	An optional attribute that specifies how string data is passed in the message. <code>false</code> specifies that string data is passed as <code>TIBMSG_STRING</code> . <code>true</code> specifies that string data is passed as <code>OPAQUE</code> . This value will override the value set in <code>tibrv:binding</code> .

In addition to the above properties, `tibrv:input` can also specify a policy for how array data is handled for messages using the binding. The array policy is set using a child `tibrv:array` element. The array policy set at this level overrides any policies set at the binding level or the operation level. For information on defining array policies see [“Defining Array Mapping Policies” on page 113](#).

The `tibrv:input` element also defines any custom mappings between the WSDL messages defined in the contract and the physical `TibrvMsg` on the wire. A custom mapping can also include binding-only message data and context information. For information on defining custom data mappings see [“Defining a Custom `TibrvMsg` Mapping” on page 119](#).

tibrv:output

`tibrv:output` is the immediate child of a WSDL `output` element and defines a number of properties used in mapping the output message to a `TibrvMsg`. Its attributes are described in [Table 13](#).

Table 14: *Attributes for `tibrv:output`*

Attribute	Purpose
<code>messageNameFieldPath</code>	An optional attribute that specifies the field path that includes the message name. If this attribute is not specified, the first field in the top level message will be used as the message name and given the value <code>IT_BUS_MESSAGE_NAME</code> .

Table 14: *Attributes for tibrv:output*

Attribute	Purpose
messageNameFieldValue	An optional attribute that specifies the field value that corresponds to the message name. If this attribute is not specified, the WSDL message's name will be used.
stringEncoding	An optional attribute that specifies the character set used in encoding string data included in the message. This value will override the value set in tibrv:binding .
stringAsOpaque	An optional attribute that specifies how string data is passed in the message. <code>false</code> specifies that string data is passed as <code>TIBRMSG_STRING</code> . <code>true</code> specifies that string data is passed as <code>OPAQUE</code> . This value will override the value set in tibrv:binding .

In addition to the above properties, `tibrv:output` can also specify a policy for how array data is handled for messages using the binding. The array policy is set using a child `tibrv:array` element. The array policy set at this level overrides any policies set at the binding level or the operation level. For information on defining array policies see [“Defining Array Mapping Policies” on page 113](#).

The `tibrv:output` element also defines any custom mappings between the WSDL messages defined in the contract and the physical TibrvMsg on the wire. A custom mapping can also include binding-only message data and context information. For information on defining custom data mappings see [“Defining a Custom TibrvMsg Mapping” on page 119](#).

Example

[Example 55](#) shows an example of an Artix contract containing a default TibrvMsg binding.

Example 55: *Default TibrvMsg Binding*

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

Example 55: *Default TibrvMsg Binding*

```

<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tibrv="http://schemas.iona.com/transport/tibrv"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:complexType name="Address">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street" type="xsd:string" minOccurs="1" maxOccurs="5"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zipCode" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="widgetOrderInfo">
        <xsd:sequence>
          <xsd:element name="amount" type="xsd:int"/>
          <xsd:element name="order_date" type="xsd:string"/>
          <xsd:element name="type" type="xsd:string"/>
          <xsd:element name="shippingAddress" type="xsd1:Address"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="widgetOrderBillInfo">
        <xsd:sequence>
          <xsd:element name="amount" type="xsd:int"/>
          <xsd:element name="order_date" type="xsd:string"/>
          <xsd:element name="type" type="xsd:string"/>
          <xsd:element name="amtDue" type="xsd:float"/>
          <xsd:element name="orderNumber" type="xsd:string"/>
          <xsd:element name="shippingAddress" type="xsd1:Address"/>
        </xsd:sequence>
      </xsd:complexType>
    </schema>
  </types>
  <message name="widgetOrder">
    <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
  </message>

```

Example 55: *Default TibrvMsg Binding*

```
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <tibrv:binding/>
  <operation name="placeWidgetOrder">
    <tibrv:operation/>
    <input name="widgetOrder">
      <tibrv:input/>
    </input>
    <output name="widgetOrderBill">
      <tibrv:output/>
    </output>
  </operation>
</binding>
<service name="orderWidgetsService">
  <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
    ...
  </port>
</service>
</definitions>
```

Artix Default Mappings for TibrvMsg

TIBRVMSG type mapping

Table 15 shows how Artix maps XSD types into TibrvMsg data types.

Table 15: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD
TIBRVMSG_STRING	xsd:string
TIBRVMSG_BOOL	xsd:boolean
TIBRVMSG_I8	xsd:byte
TIBRVMSG_I16	xsd:short
TIBRVMSG_I32	xsd:int
TIBRVMSG_I64	xsd:long
TIBRVMSG_U8	xsd:unsignedByte
TIBRVMSG_U16	xsd:unsignedShort
TIBRVMSG_U32	xsd:unsignedInt
TIBRVMSG_U64	xsd:unsignedLong
TIBRVMSG_F32	xsd:float
TIBRVMSG_F64	xsd:double
TIBRVMSG_STRING	xsd:decimal
TIBRVMSG_DATETIME ^a	xsd:dateTime
TIBRVMSG_OPAQUE	xsd:base64Binary
TIBRVMSG_OPAQUE	xsd:hexBinary
TIBRVMSG_STRING	xsd:QName
TIBRVMSG_STRING	xsd:nonPositiveInteger
TIBRVMSG_STRING	xsd:negativeInteger

Table 15: TIBCO to XSD Type Mapping

TIBRVMSG	XSD
TIBRVMSG_STRING	xsd:nonNegativeInteger
TIBRVMSG_STRING	xsd:positiveInteger
TIBRVMSG_STRING	xsd:time
TIBRVMSG_STRING	xsd:date
TIBRVMSG_STRING	xsd:gYearMonth
TIBRVMSG_STRING	xsd:gMonthDay
TIBRVMSG_STRING	xsd:gDay
TIBRVMSG_STRING	xsd:gMonth
TIBRVMSG_STRING	xsd:anyURI
TIBRVMSG_STRING	xsd:token
TIBRVMSG_STRING	xsd:language
TIBRVMSG_STRING	xsd:NMTOKEN
TIBRVMSG_STRING	xsd:Name
TIBRVMSG_STRING	xsd:NCName
TIBRVMSG_STRING	xsd:ID

a. While TIBRVMSG_DATETIME has microsecond precision, xsd:dateTime only supports millisecond precision. Therefore, Artix rounds all times to the nearest millisecond.

Sequence complex types

Sequence complex types are mapped to a TibrvMsg message as follows:

- The elements of the complex type are enclosed in a TibrvMsg instance.
- If the complex type is specified as a message part, the value of the `part` element's `name` attribute is used as the name of the generated TibrvMsg.
- If the complex type is specified as an element, the value of the `element` element's `name` attribute is used as the name of the generated TibrvMsg.

- The `TibrvMsg` id is 0.
 - The elements are mapped to child `TibrvMsgField` instances of the wrapping `TibrvMsg`.
 - If an element of the sequence is of a complex type, it will be mapped into a `TibrvMsg` instance that conforms to the default mapping rules.
 - The value of the `element` element's `name` attribute is used as the name of the generated `TibrvMsgField` instance.
 - The child fields' ids are 0.
 - The child fields are serialized in the same order as they appear in the schema definition.
 - The child fields are deserialized in the same order as they appear in schema definition.
-

All complex types

All complex types are mapped to a `TibrvMsg` message as follows:

- The elements of the complex type are enclosed in a `TibrvMsg` instance.
 - If the complex type is specified as a message part, the value of the `part` element's `name` attribute is used as the name of the generated `TibrvMsg`.
 - If the complex type is specified as an element, the value of the `element` element's `name` attribute is used as the name of the generated `TibrvMsg`.
 - The `TibrvMsg` id is 0.
 - The elements are mapped to child `TibrvMsgField` instances of the wrapping `TibrvMsg`.
 - If an element of the all is of a complex type, it will be mapped into a `TibrvMsg` instance that conforms to the default mapping rules.
 - The value of the `element` element's `name` attribute is used as the name of the generated `TibrvMsgField` instance.
 - The child field's ids are 0.
 - The child fields are serialized in the same order as they appear in the schema definition.
 - The child fields can be deserialized in any order.
-

Choice complex types

Choice complex types are mapped to a `TibrvMsg` message as follows:

- The elements of the complex type are enclosed in a TibrvMsg instance.
- If the complex type is specified as a message part, the value of the `part` element's `name` attribute is used as the name of the generated TibrvMsg.
- If the complex type is specified as an element, the value of the `element` element's `name` attribute is used as the name of the generated TibrvMsg.
- The TibrvMsg id is 0
- There must only be one and only child field in this TibrvMsg message that corresponds to the active choice element.
- The child field has the name of the corresponding choice's active element name.
- The child field id is zero.
- During deserialization the binding runtime will extract the first child field from this message using index equal to 0 as the key. If no field is found then this choice is considered an empty choice.

NMTOKEN

NMTOKEN schema types are mapped as follows:

- The NMTOKEN is enclosed in a TibrvMsg instance.
- If the NMTOKEN is specified as a message part, the value of the `part` element's `name` attribute is used as the name of the generated TibrvMsg.
- If the NMTOKEN is specified as an element, the value of the `element` element's `name` attribute is used as the name of the generated TibrvMsg.
- The TibrvMsg id is 0
- Each NMTOKEN is mapped to a child TibrvMsgField instance of this TibrvMsg.
- The names of the children fields are an ever increasing counter values beginning with 0.

Default mapping of arrays

XML Schema elements that are not mapped to native Tibrv scalar types and have `minOccurs != 1` and `maxOccurs != 1` are mapped as follows:

- Array elements are stored in a TibrvMsg instance at the same scope as the sibling elements of this array element.

- Array element names are a result of an expression evaluation. The expression is evaluated for every array element.
- The default array element name expression is `concat(xml:attr('name'), '_', counter(1, 1))`.
- If an instance of an array element has 0 elements then this array instance will have nothing loaded onto the wire. Currently this is not true for scalar arrays that are loaded as a single field.

Default mapping for scalar arrays

The XML Schema elements that are mapped to native Tibrv scalar types and have `minOccurs != 1` and `maxOccurs != 1` are mapped as follows:

- Array elements are stored in a `TibrvMsg` instance at the same scope as sibling elements of this array element.
- The binding utilizes the Tibrv native array mapping to store XML Schema arrays. Hence, there will be only one `TibrvMsgField` with the name equal to that of the XML Schema element name defining this array.

Defining Array Mapping Policies

Overview

Because `TibrvMsg` does not natively support sparsely populated arrays, the Artix `TibrvMsg` binding allows you to define how array elements are mapped into a `TibrvMsg` when they are written to the wire using the `tibrv:array` element. In addition, the Artix `TibrvMsg` binding allows you to define the naming schema used for array elements when they are mapped into `TibrvMsgField` instances.

Policy scoping

The `tibrv:array` element can define array properties at any level of granularity by making it the child of different `TibrvMsg` binding elements. [Table 16](#) shows the effect of setting `tibrv:array` at different levels of a binding.

Table 16: *Effect of `tibrv:array`*

Child of	Effect
<code>tibrv:binding</code>	Sets the array policies for all messages in the binding.
<code>tibrv:operation</code>	Array policies set at the operation level only affect the messages defined within the parent <code>operation</code> element. They override any array policies set at the binding level.
<code>tibrv:input</code>	Array policies set at this level only affect the input message. They override any array policies set at the binding or operation level.
<code>tibrv:output</code>	Array policies set at this level only affect the output message. They override any array policies set at the binding or operation level.
<code>tibrv:msg</code>	Array policies set at this level affect only the fields defined within the <code>tibrv:msg</code> element. They override any array policies set at higher levels.

Table 16: *Effect of tibrv:array*

Child of	Effect
tibrv:field	Array policies set at this level affect only the TibrvMsg field being defined. They override any array policies set at higher levels.

Array policies

The array policies are set using the attributes of `tibrv:array`. [Table 17](#) describes the attributes used to set array policies.

Table 17: *Attributes for tibrv:array*

Attribute	Purpose
elementName	Specifies an expression that when evaluated will be used as the name of the TibrvMsg field to which array elements are mapped. The default element naming scheme is to concatenate the value of WSDL <code>element</code> element's <code>name</code> attribute with a counter. For information on specifying naming expressions see “Custom array naming expressions” on page 116 .
integralAsSingleField	Specifies how scalar array data is mapped into TibrvMsgField instances. <code>true</code> , the default, specifies that arrays are mapped into a single TibrvMsgField. <code>false</code> specifies that each member of an array is mapped into a separate TibrvMsgField.
loadSize	Specifies if the number of elements in an array is included in the TibrvMsg. <code>true</code> specifies that the number of elements in the array is added as a TibrvMsgField in the same TibrvMsg as the array. <code>false</code> , the default, specifies that the number of elements in the array is not included in the TibrvMsg.

Table 17: *Attributes for tibrv:array*

Attribute	Purpose
sizeName	Specifies an expression that when evaluated will be used as the name of the TibrvMsgField to which the size of the array is written. The default naming scheme is to concatenate the value of WSDL <code>element</code> element's <code>name</code> attribute with <code>@size</code> . For information on specifying naming expressions see "Custom array naming expressions" on page 116.

Sparse arrays

A sparse array is an array with some of the elements set to nil. For instance, if an array has 10 elements, the 3rd and fifth elements may be nil. Tibco/RV has no way of natively representing sparse arrays or nil element members. This presents two problems:

- Tibco/RV throws an exception when it encounters nil scalar values that are mapped to a TibrvMsgField.
- There is no mechanism for maintaining the element positions of the non-nil members of the array.

To solve both problems you would specify array policies such that the size of the array is written to the wire and that each element of the array is written to the wire as a separate TibrvMsgField. To specify that the array size is written to the wire use `loadSize="true"`. To specify that each member of the array is written in a separate TibrvMsgField use `integralAsSingleField="false"`.

The resulting TibrvMsg would have one field for each non-nil member of the array and a field specifying the size of the array. Artix can use this information to reconstruct the sparse array when it is passed through the TibrvMsg binding. A Tibco/RV application would need to implement the logic to handle the information.

Custom array naming expressions When specifying a naming policy for array element names you use a string expression that combines XML properties, strings, and custom naming functions. For example, you could use the expression `concat(xml:attr('name'), '_', counter(1,1))` to specify that each element in the array `street` is named `street_n`.

[Table 18](#) shows the available functions for use in building array element names.

Table 18: *Functions Used for Specifying TibrvMsg Array Element Names*

Function	Purpose
<code>xml:attr('attribute')</code>	Inserts the value of the named attribute.
<code>concat(item1, item2, ...)</code>	Concatenates all of the elements into a single string.
<code>counter(start, increment)</code>	Adds an increasing numerical value. The counter starts at <i>start</i> and increases by <i>increment</i> .

Example

[Example 56](#) shows an example of an Artix contract containing a TibrvMsg binding that uses array policies. The policies are set at the binding level and:

- Force the name of the TibrvMsg containing array elements to be named `street0`, `street1`, ...
- Write out the number of elements in each `street` array.
- Force each element of a `street` array to be written out as a separate field.

Example 56: *TibrvMsg Binding with Array Policies Set*

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

Example 56: *TibrVMsg Binding with Array Policies Set*

```

<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tibrv="http://schemas.iona.com/transport/tibrv"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:complexType name="Address">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street" type="xsd:string" minOccurs="1" maxOccurs="5"
            nillable="true"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zipCode" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </schema>
  </types>
  <message name="addressRequest">
    <part name="resident" type="xsd:string"/>
  </message>
  <message name="addressResponse">
    <part name="address" type="xsd1:Address"/>
  </message>
  <portType name="theFourOneOne">
    <operation name="lookUp">
      <input message="tns:addressRequest" name="request"/>
      <output message="tns:addressResponse" name="response"/>
    </operation>
  </portType>

```

Example 56: *TibrvMsg Binding with Array Policies Set*

```

<binding name="lookUpBinding" type="tns:theFourOneOne">
  <tibrv:binding>
    <tibrv:array elementName="concat(xml:attr('name'), counter(0, 1))"
      integralsAsSingleField="false"
      loadSize="true"/>
  </tibrv:binding>
  <operation name="lookUp">
    <tibrv:operation/>
    <input name="addressRequest">
      <tibrv:input/>
    </input>
    <output name="addressResponse">
      <tibrv:output/>
    </output>
  </operation>
</binding>
<service name="orderWidgetsService">
  <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
    ...
  </port>
</service>
</definitions>

```

Defining a Custom TibrvMsg Mapping

Overview

For instances where the default mappings are insufficient to map the TibrvMsgs to corresponding WSDL messages, you can define custom mappings that allow you to specify exactly how the WSDL message parts are mapped into a TibrvMsg. Custom TibrvMsg mappings allow you to:

- override the native XML Schema type specification of contract elements.
- add binding-only elements to the TibrvMsg placed on the wire.
- place globally used contract elements in higher levels of the binding.
- change how contract elements are mapped into nested TibrvMsg structures.

Custom TibrvMsg binding elements are defined using a combination of [tibrv:msg](#) elements and [tibrv:field](#) elements.

Contract elements vs. binding-only elements

A *contract element* is an atomic piece of a message defined in the logical description of the interface being bound. It can be a native XML Schema type such as `xsd:int`, in which case it is mapped to a TibrvMsgField. Or it can be an instance of a complex type, in which case it is mapped to a TibrvMsg. For example, if a message has a part that is of type `xsd:string`, the part is a contract element. In contract fragment shown in [Example 57](#),

the message part `title` is a contract element that will be mapped to a `TibrvMsgField`. The message part `tale` is a contract element that will be mapped to a `TibrvMsg` that contains three `TibrvMsgField` entries.

Example 57: *TibrvMsg Contract Elements*

```
<types>
  ...
  <complexType name="leda">
    <sequence>
      <element name="castor" type="xsd:string"/>
      <element name="pollux" type="xsd:string"/>
      <element name="hellen" type="xsd:boolean"/>
    </sequence>
  </complexType>
  ...
</types>
<message name="taleRequest">
  <part name="title" type="xsd:string"/>
</message>
<message name="taleResponse">
  <part name="tale" type="xsd1:leda"/>
</message>
```

A *binding-only element* is any artifact that is added to the message as part of the binding. The main purpose of a binding-only element is to add data required by a native Tibco application to a message produced by an Artix application. Binding-only elements are not passed back into an Artix application. However, a native Tibco application will have access to binding-only elements.

Scoping

You can add custom `TibrvMsg` binding elements to any of the `TibrvMsg` binding elements. The order in which custom `TibrvMsg` binding elements are serialized is as follows:

1. Immutable root `TibrvMsg` wrapper.
2. Custom elements defined in `tibrv:binding` are added for all messages.
3. Custom elements defined in `tibrv:operation` for all messages used by the WSDL operation.
4. Custom elements defined in `tibrv:input` or `tibrv:output` for the specific message.

If you define a binding-only element in the `tibrv:binding` element, it will be the first field in the `TibrvMsg` generated for all messages that are generated by the binding. If you also added a binding-only field in the `tibrv:operation` for the operation `getHeader`, messages used by `getHeader` would have both binding-only fields.

Note: If you add a custom mapped contract element at any scope above the `tibrv:input` or the `tibrv:output` level, you must be certain that it is part of the logical messages for all elements at a lower scope. For example, if a contract element is given a custom mapping in a `tibrv:operation`, the corresponding WSDL message must be used by both the input and output messages. If it is not an exception will be thrown.

Casting XMLSchema types

If the default mapping between the type of a contract element and the type of the corresponding `TibrvMsgField` is not appropriate, you can use the `type` attribute of `tibrv:field` to change the type of the contract element. The `type` attribute allows you to cast one native XML Schema type into another native XML Schema type.

When Artix finds a `tibrv:field` element whose name attribute corresponds to a `part` defined in the contract, or an `element` of a complex type used as a `part`, and whose `type` attribute is set, it will convert the value of the message part into the specified type. For example, given the contract fragment in [Example 58](#), the value of `casted` would be converted from an `int` to a `string`. So if `casted` had a value of `3`, the `TibrvMsg` binding would turn it into the string `'3'`.

Example 58: Casting in a TibrvMsg Binding

```
<definitions ...>
  ...
  <message name="request">
    <part name="input1" type="xsd:int"/>
  </message>
  <portType name="castor">
    <operation name="ascend">
      <input message="tns:request" name="day"/>
    </operation>
    ...
  </portType>
```

Example 58: *Casting in a TibrvMsg Binding*

```

<binding name="castorTib" portType="castor">
  <tibrv:binding/>
  <operation name="ascend">
    <tibrv:operation/>
    <input message="tns:request" name="day">
      <tibrv:input>
        <tibrv:field name="input1" type="xsd:string"/>
      </tibrv:input>
    </input>
  </operation>
  ...
</binding>
...
</definitions>

```

Table 19 shows the matrix of valid casts for native XML Schema types.

Table 19: *Valid Casts for TibrvMsg Binding*

Type	Full Support	Restricted Support ^a
byte	short, int, long, float, double, decimal, string, boolean	unsignedByte, unsignedShort, unsignedInt, unsignedLong
unsignedByte	short, unsignedShort, int, unsignedInt, long, unsignedLong, float, double, decimal, string, boolean	byte
short	int, long, float, double, decimal, string, boolean	byte, unsignedByte, unsignedShort, unsignedInt, unsignedLong
unsignedShort	byte, unsignedByte, short	int, unsignedInt, long, unsignedLong, float, double, decimal, string, boolean

Table 19: *Valid Casts for TibrvMsg Binding*

Type	Full Support	Restricted Support ^a
int	long, decimal, string, boolean	byte, unsignedByte, short, unsignedShort, unsignedInt, unsignedLong, float, double
unsignedInt	long, unsignedLong, decimal, string, boolean	byte, unsignedByte, short, unsignedShort, int, float, double
long	decimal, string, boolean	byte, unsignedByte, short, unsignedShort, int, unsignedInt, unsignedLong, float, double
unsignedLong	decimal, string, boolean	byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, float, double
float	double, decimal, string, boolean	byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, unsignedLong
double	decimal, string, boolean	byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, unsignedLong, float
decimal	string, boolean	byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, unsignedLong, float, double

Table 19: *Valid Casts for TibrvMsg Binding*

Type	Full Support	Restricted Support ^a
string ^b		byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, unsignedLong, float, double, decimal, boolean, QName, DateTime
boolean	byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, unsignedLong, float, double	decimal, string
QName		string
DateTime		string

a. Must be within the appropriate value range.

b. In addition to a, the syntax must also conform

Adding binding-only elements to a contract

As mentioned in [“Scoping” on page 120](#), a binding-only element can be added to a TibrvMsg binding at any point in its definition. Before adding a binding-only element you should determine the proper placement for its inclusion in the binding. For example, if you are interoperating with a Tibco system that expects every message to have a header, you would most likely add the header definition in the `tibrv:binding` element.

However, if the Tibco system required a static footer for every message, you would need to add the footer to the `tibrv:input` and `tibrv:output` elements. This is because of the serialization order of the elements in the TibrvMsg binding. Elements are added to the serialized message from the global scope to the local scope in order.

Binding-only elements are specified using a combination of `tibrv:msg` elements and `tibrv:field` elements. When specifying a binding-only element you need to specify a value for the `alias` attribute. The `alias` attribute specifies the name of the generated TibrvMsg element. For `tibrv:field` elements you also need to specify values for the `type` attribute and the

value attribute. The `type` attribute specifies the XML Schema type of the element being added and the `value` attribute specifies the value to be placed in the resulting `TibrvMsgField`.

[Example 59](#) shows a `TibrvMsg` binding that adds a static header to each message that is put on the wire.

Example 59: *TibrvMsg Binding with Binding-only Elements*

```
<binding name="headedTibcoBinding" portType="mythMaker">
  <tibrv:binding>
    <tibrv:msg alias="header">
      <tibrv:field alias="class" type="xs:string" value="greek"/>
      <tibrv:field alias="form" type="xs:string" value="poetry"/>
    </tibrv:msg>
  </tibrv:binding>
  <operation name="spinner">
    ...
  </operation>
  ...
</binding>
```

A message generated by the binding in [Example 59](#) would have as its first member a `TibrvMsg` called `header` as shown in [Example 60](#).

Example 60: *TibrvMsg with a Header*

```
TibrvMsg
{
  TibrvMsgField
  {
    name = "header";
    id = 0;
    data.msg =
    {
      TibrvMsgField
      {
        name = "class";
        id = 0;
        data.str = "greek";
        size = sizeof(data);
        count = 1;
        type = TIBRVMSG_STRING;
      }
    }
  }
}
```

Example 60: *TibrvMsg with a Header*

```

TibrvMsgField
{
    name = "form";
    id = 0;
    data.str = "poetry";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_STRING;
}
}
size = sizeof(data);
count = 1;
type = TIBRVMSG_MSG;
}
...
}

```

Placing binding-only elements between contract elements

In addition to adding extra-information at the beginning and end of messages, you can place binding-only elements between contract elements in a message. For example, the default mapping of the message `taleResponse`, defined in [Example 57](#), would produce the `TibrvMsg` shown in [Example 61](#).

Example 61: *Default TibrvMsg Example*

```

TibrvMsg
{
    TibrvMsgField
    {
        name = "tale";
        id = 0;
        data.msg =
        {
            TibrvMsgField
            {
                name = "castor";
                id = 0;
                data.str = "This one is a horse trainer.";
                size = sizeof(data);
                count = 1;
                type = TIBRVMSG_STRING;
            }
        }
    }
}

```

Example 61: *Default TibrvMsg Example*

```

TibrvMsgField
{
    name = "pollux";
    id = 0;
    data.str = "This one is a boxxer.";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_STRING;
}
TibrvMsgField
{
    name = "hellen";
    id = 0;
    data.str = "false";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_BOOL;
}
}
size = sizeof(data);
count = 1;
type = TIBRVMSG_MSG;
}
...
}

```

If the Tibco application you are integrating with requires an additional TibrvMsgField or an additional TibrvMsg between `pollux` and `hellen`, as shown in [Example 62](#), you could add it to the binding by redefining the mapping of the entire contract element to include a binding-only element.

Example 62: *TibrvMsg with added TibrvMsg Example*

```

TibrvMsg
{
    TibrvMsgField
    {
        name = "tale";
        id = 0;
        data.msg =
        {

```

Example 62: *TibrvMsg with added TibrvMsg Example*

```

TibrvMsgField
{
    name = "castor";
    id = 0;
    data.str = "This one is a horse trainer.";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_STRING;
}
TibrvMsgField
{
    name = "pollux";
    id = 0;
    data.str = "This one is a boxxer.";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_STRING;
}
TibrvMsgField
{
    name = "clytemnestra";
    id = 0;
    data.msg =
    {
        TibrvMsgField
        {
            name = "father";
            id = 0;
            data.str = "tyndareus";
            size = sizeof(data);
            count = 1;
            type = TIBRVMSG_STRING;
        }
        TibrvMsgField
        {
            name = "husbands";
            id = 0;
            data.i32 = 2;
            size = sizeof(data);
            count = 1;
            type = TIBRVMSG_I32;
        }
    }
}

```

Example 62: *TibrvMsg with added TibrvMsg Example*

```

    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_MSG;
}
TibrvMsgField
{
    name = "hellen";
    id = 0;
    data.str = "false";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_BOOL;
}
}
size = sizeof(data);
count = 1;
type = TIBRVMSG_MSG;
}
...
}

```

To add the binding-only element `clytemnestra` to the default binding of the message `leda`:

1. Because the message `leda` is used as an output message, add a `tibrv:msg` child element to the `tibrv:output` element.
2. Set the `tibrv:msg` element's `name` attribute to the value of the corresponding contract message part that uses the type `leda`.
3. Add a `tibrv:field` element as a child of the `tibrv:msg` element.
4. Set the new `tibrv:field` element's `name` attribute to the value of the corresponding element's `name` attribute. In this instance, `castor`.
5. Repeat steps **3** and **4** for the second element, `pollux`, in `leda`.
6. To start the binding-only TibrvMsg element, add a `tibrv:msg` element after the `tibrv:field` element for `pollux`.
7. Set the new `tibrv:msg` element's `alias` attribute to `clytemnestra`.
8. Add a `tibrv:field` element as a child of the `tibrv:msg` element.
9. Set the `tibrv:field` element's `alias` attribute to `father`.
10. Set the `tibrv:field` element's `type` attribute to `xsd:string`.
11. Set the `tibrv:field` element's `value` attribute to `tyndareus`.

12. Repeat steps 8 through 11 for the second `TibrvMsgField` in `clytemnestra`.
13. On the same level as the `tibrv:field` elements mapping `castor` and `pollux`, add a `tibrv:field` element to map `helen`.

[Example 63](#) shows a binding for the message shown in [Example 62](#).

Example 63: *TibrvMsg Binding with an Added Binding-only Element*

```
<binding name="tibBinding">
  <tibrv:binding/>
  <operation ...>
    <tibrv:operation/>
    <input ...>
      <tibrv:input/>
    </input>
    <output name="response" message="tns:taleResponse">
      <tibrv:output>
        <tibrv:msg name="tale">
          <tibrv:field name="castor"/>
          <tibrv:field name="pollux"/>
          <tibrv:msg alias="clytemnestra">
            <tibrv:field alias="father" type="xsd:string"
              value="tyndareus"/>
            <tibrv:field alias="husbands" type="xsd:int"
              value="2"/>
          </tibrv:msg/>
          <tibrv:field name="hellen"/>
        </tibrv:msg>
      </tibrv:output>
    </output>
  </operation>
</binding>
```

Creating a custom mapping for a message defined in the contract

Using the `tibrv:msg` elements and `tibrv:field` elements you can change how contract elements are broken into `TibrvMsgs` and `TibrvMsgFields`. For a detailed discussion of the default `TibrvMsg` mapping see [“Artix Default Mappings for TibrvMsg”](#) on page 108.

You can alter this default mapping to add more wrapping to the TibrvMsgFields. For instance, if a message consists of a single `xsd:string` part, it would be mapped to a TibrvMsg similar to the one shown in [Example 64](#).

Example 64: *TibrvMsg for a String*

```
TibrvMsg
{
  TibrvMsgField
  {
    name = "electra";
    id = 0;
    data.str = "forelorn";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_STRING;
  }
}
```

However, you could specify that instead of being mapped straight to a TibrvMsgField, it be mapped to a TibrvMsg containing a TibrvMsgField as shown in [Example 65](#).

Example 65: *TibrvMsg with a TibrvMsg with a String*

```
TibrvMsg
{
  TibrvMsgField
  {
    name = "grandchild";
    id = 0;
    data.msg =
```

Example 65: *TibrvMsg with a TibrvMsg with a String*

```

{
  TibrvMsgField
  {
    name = "electra";
    id = 0;
    data.str = "forelorn";
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_STRING;
  }
  size = sizeof(data);
  count = 1;
  type = TIBRVMSG_MSG;
}
}

```

To increase the depth of the wrapping of contract elements you define a custom TibrvMsg mapping that adds the desired number of levels. Each new level of wrapping is specified by a `tibrv:msg` element. To create the message shown in [Example 65](#) you would use a binding definition similar to the one shown in [Example 66](#).

Example 66: *TibrvMsg Binding with an Extra TibrvMsg Level*

```

<binding name="tibBinding">
  <tibrv:binding/>
  <operation ...>
    <tibrv:operation/>
    <input ...>
      <tibrv:input>
        <tibrv:msg alias="gradnchild">
          <tibrv:field name="electra" type="xsd:string"/>
        </tibrv:msg>
      </input>
    </operation>
  </binding>

```

You can also use this feature to alter the wrapping of complex type elements. For example, if you were using the message defined in [Example 57](#) the default TibrvMsg would consist of one TibrvMsg, `leda`, containing 3 fields, one for each element in the structure, wrapped by the

root TibrvMsg. You could modify the mapping of the logical message to a TibrvMsg that resembles the one shown in [Example 67](#). The two elements `castor` and `pollux` have been wrapped in a TibrvMsg called `brothers`.

Example 67: *TibrvMsg with Custom TibrvMsg Wrapping*

```
TibrvMsg
{
  TibrvMsgField
  {
    name = "tale";
    id = 0;
    data.msg =
    {
      TibrvMsgField
      {
        name = "brothers"
        id = 0;
        data.msg =
        {
          TibrvMsgField
          {
            name = "castor";
            id = 0;
            data.str = "This one is a horse trainer.";
            size = sizeof(data);
            count = 1;
            type = TIBRVMSG_STRING;
          }
          TibrvMsgField
          {
            name = "pollux";
            id = 0;
            data.str = "This one is a boxxer.";
            size = sizeof(data);
            count = 1;
            type = TIBRVMSG_STRING;
          }
        }
      }
    }
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_MSG;
  }
}
```

Example 67: *TibrvMsg with Custom TibrvMsg Wrapping (Continued)*

```

TibrvMsgField
{
    name = "hellen";
    id = 0;
    data.bool = false;
    size = sizeof(data);
    count = 1;
    type = TIBRVMSG_BOOL;
}
}
size = sizeof(data);
count = 1;
type = TIBRVMSG_MSG;
}
...
}

```

Adding additional levels of wrapping within a complex type is done the same way as it is done with a message part. You place additional `tibrv:msg` elements around the contract elements you want to be at a deeper level. [Example 68](#) shows a binding fragment that would create the `TibrvMsg` shown in [Example 67](#).

Example 68: *Binding of a Complex Type with an Extra TibrvMsg Level*

```

<binding name="tibBinding">
  <tibrv:binding/>
  <operation ...>
    <tibrv:operation/>
    <input ...>
      <tibrv:input>
        <tibrv:msg name="tale">
          <tibrv:msg alais="brothers">
            <tibrv:field name="castor" type="xsd:string"/>
            <tibrv:field name="pollux" type="xsd:string"/>
          </tibrv:msg>
          <tibrv:field name="hellen" type="xsd:boolean"/>
        </tibrv:msg>
      </tibrv:input>
    </input>
    ...
  </operation>
</binding>

```

tibrv:msg

`tibrv:msg` instructs the binding runtime to create an instance of a `TibrvMsg`. Its attributes are described in [Table 20](#).

Table 20: *Attributes for tibrv:msg*

Attribute	Purpose
<code>name</code>	Specifies the name of the contract element from which this <code>TibrvMsg</code> instance gets its value. If this attribute is not present, then the <code>TibrvMsg</code> is considered a binding-only element.
<code>alias</code>	Specifies the value of the <code>name</code> member of the <code>TibrvMsg</code> instance. If this attribute is not specified, then the binding will use the value of the <code>name</code> attribute.
<code>element</code>	Used only when <code>tibrv:msg</code> is an immediate child of <code>tibrv:context</code> . Specifies the QName of the element defining the context data to use when populating the <code>TibrvMsg</code> . See “Adding Context Information to a TibrvMsg” on page 137 .
<code>id</code>	Specifies the value of the <code>id</code> member of the <code>TibrvMsg</code> instance. The default value is <code>0</code> .
<code>minOccurs/</code> <code>maxOccurs</code>	Used only with contract elements. The values must be identical to the values specified in the schema definition.

tibrv:field

`tibrv:field` instructs the binding to create an instance of a `TibrvMsgField`. Its attributes are described in [Table 21](#).

Table 21: *Attributes for tibrv:field*

Attribute	Purpose
<code>name</code>	Specifies the name of the contract element from which this <code>TibrvMsgField</code> instance gets its value. If this attribute is not present, then the <code>TibrvMsgField</code> is considered a binding-only element.

Table 21: *Attributes for `tibrv:field`*

Attribute	Purpose
alias	Specifies the value of the <code>name</code> member of the <code>TibrvMsgField</code> instance. If this attribute is not specified, then the binding will use the value of the <code>name</code> attribute.
element	Used only when <code>tibrv:field</code> is an immediate child of <code>tibrv:context</code> . Specifies the QName of the element defining the context data to use when populating the <code>TibrvMsgField</code> . See “Adding Context Information to a <code>TibrvMsg</code>” on page 137 .
id	Specifies the value of the <code>id</code> member of the <code>TibrvMsgField</code> instance. The default value is <code>o</code> .
type	Specifies the XML Schema type of the data being used to populate the <code>data</code> member of the <code>TibrvMsgField</code> instance. For a list of supported types, see “Artix Default Mappings for <code>TibrvMsg</code>” on page 108 .
value	Specifies the value inserted into the <code>data</code> member of the <code>TibrvMsgField</code> instance when the field is a binding-only element.
minOccurs/ maxOccurs	Used only with contract elements. The values must be identical to the values specified in the schema definition.

Adding Context Information to a TibrvMsg

Overview

By using Artix contexts, you can define binding-only data that is dynamically generated and consumed by Artix applications. Contexts are a feature of the Artix programming model that allow application developers to pass metadata up and down the messaging chain. When using the TibrvMsg binding, you can instruct your Artix application to use context data to populate outgoing binding-only fields. On the receiving end, the TibrvMsg binding takes the information and uses it to populate a context in the application. For information on using contexts in Artix applications, see [Developing Artix Applications with C++](#) or [Developing Artix Applications with Java](#).

Telling the binding to get information from Artix contexts

When defining a custom TibrvMsg binding, you use the `tibrv:context` element to inform the binding that the immediate child element is populated from an Artix context. The immediate child of a `tibrv:context` element must be either a `tibrv:msg` element or a `tibrv:field` element depending on what type of information is contained in the context.

You would use `tibrv:msg` for context data that is an instance of a complex XML Schema type. You could also use `tibrv:msg` if you want an instance of a native XML Schema type wrapped in a TibrvMsg. You would use `tibrv:field` to insert context data that was an instance of a native XML Schema type as a TibrvMsgField.

When a `tibrv:msg` element or a `tibrv:field` element are used to insert context information into a TibrvMsg they use the `element` attribute in place of the `name` attribute. The `element` attribute specifies the QName used to register the context data with the Artix bus. It must correspond to a globally defined XML Schema element. Also, when inserting context information you cannot specify values for any other attributes except the `alias` attribute.

Application considerations

When using context data in your TibrvMsg binding there is some application-specific information you need to abide by:

- At least one piece of the integrated solution must be an Artix application to process the context data.

- The Tibrv binding will automatically register, but not create an instance of, any contexts used in its binding definition with the Artix bus. Contexts are registered using the QName of the element specified in the contract.
- For any context data that will be sent in an input message, client-side Artix applications are responsible for creating an instance of the appropriate context data in the request context container before the message is handed off to the binding.
- Context data sent from a client in an input message will be available to server-side Artix applications in the request context once the message has been processed by the binding.
- For any context data that will be sent in an output message, server-side Artix applications are responsible for creating an instance of the appropriate context data in the reply context container before the message is handed off to the binding.
- Context data sent from a server in an output message will be available to client-side Artix applications in the reply context once the message has been processed by the binding.

Example

If you were integrating with a Tibco server that used a header to correlate messages using an ASCII correlation ID, you could use the TibrvMsg binding's context support to implement the correlation ID on the Artix side of the solution. The first step would be to define an XML Schema element called `corrID` for the context that would hold the correlation ID. Then in your TibrvMsg binding definition you would include a `tibrv:context` element in the `tibrv:binding` element to specify that all messages passing through the binding will have the header. [Example 69](#) shows a contract fragment containing the appropriate entries for this scenario.

Example 69: Using Context Data in a TibrvMsg Binding

```
<definitions
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  ...>
```

Example 69: *Using Context Data in a TibrvMsg Binding*

```

<types>
  <schema
    targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    ...
    <element name="corrID" type="xsd:string"/>
    ...
  </schema>
</types>
...
<portType name="correlatedService">
...
</portType>
<binding name="tibrvCorrBinding" type="correlatedService">
  <tibrv:binding>
    <tibrv:context>
      <tibrv:field element="xsd1:corrID"/>
    </tibrv:context>
  </tibrv:binding>
  ...
</binding>
...
</definitions>

```

When you develop the Artix side of the solution, you will need to supply the logic for handling the context data stored in `corrID`. The context for `corrID` will be registered with the Artix bus using the QName `"http://widgetVendor.com/types/widgetTypes", "corrID"`. If the Artix side of your solution is a client, you will need to include logic to set an appropriate `corrID` in the request context before each request and to read each response's `corrID` from the response context. If the Artix side of your application is a server, you will need to include logic to read request's `corrID` from the request context and set an appropriate `corrID` in the reply context before sending the response.

For information on using contexts in Artix applications, see [Developing Artix Applications with C++](#) or [Developing Artix Applications with Java](#).

Using XML Documents

Artix allows you to pass XML documents that are not packaged as SOAP messages.

Overview

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

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

Hand editing

To map an interface to a pure XML payload format:

1. Add the namespace declaration to include the IONA extensions defining the XML binding. See [“XML binding namespace” on page 142](#).
2. Add a standard WSDL `binding` element to your contract to hold the XML binding, give the binding a unique `name`, and specify the name of the WSDL `portType` element that represents the interface being bound.
3. Add an `xformat:binding` child element to the `binding` element to identify that the messages are being handled as pure XML documents without SOAP envelopes.

4. Optionally, set the `xformat:binding` element's `rootNode` attribute to a valid QName. For more information on the effect of the `rootNode` attribute see [“XML messages on the wire” on page 142](#).
5. For each operation defined in the bound interface, add a standard WSDL `operation` element to hold the binding information for the operation's messages.
6. For each operation added to the binding, add the `input`, `output`, and `fault` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the interface definition of the logical operation.
7. Optionally add an `xformat:body` element with a valid `rootNode` attribute to the added `input`, `output`, and `fault` elements to override the value of `rootNode` set at the binding level.

Note: If any of your messages have no parts, for example the output message for an operation that returns void, you must set the `rootNode` attribute for the message to ensure that the message written on the wire is a valid, but empty, XML document.

XML binding namespace

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

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

XML messages on the wire

When you specify that an interface's messages are to be passed as XML documents, without a SOAP envelope, you must take care to ensure that your messages form valid XML documents when they are written on the wire. You also need to ensure that non-Artix participants that receive the XML documents understand the messages generated by Artix.

A simple way to solve both problems is to use the optional `rootNode` attribute on either the global `xformat:binding` element or on the individual message's `xformat:body` elements. The `rootNode` attribute specifies the QName for the element that serves as the root node for the XML document generated by Artix. 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 rpc style messages.

For example, if the `rootNode` attribute is not set the message defined in [Example 70](#) would generate an XML document with the root element `lineNumber`.

Example 70: *Valid XML Binding Message*

```
<type ...>
  ...
  <element name="operatorID" type="xsd:int"/>
  ...
</types>
<message name="operator">
  <part name="lineNumber" element="ns1:operatorID"/>
</message>
```

For messages with one part, Artix will always generate a valid XML document even if the `rootNode` attribute is not set. However, the message in [Example 71](#) would generate an invalid XML document.

Example 71: *Invalid XML Binding Message*

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
  <element name="entryNum" type="xsd:int"/>
  ...
</types>
<message name="matildas">
  <part name="dancing" element="ns1:pairName"/>
  <part name="number" element="ns1:entryNum"/>
</message>
```

Without the `rootNode` attribute specified in the XML binding, Artix will generate an XML document similar to [Example 72](#) for the message defined in [Example 71](#). The Artix-generated XML document is invalid because it has two root elements: `pairName` and `entryNum`.

Example 72: *Invalid XML Document*

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

If you set the `rootNode` attribute, as shown in [Example 73](#) Artix will wrap the elements in the specified root element. In this example, the `rootNode` attribute is defined for the entire binding and specifies that the root element will be named `entrants`.

Example 73: *XML Format Binding with rootNode set*

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
    <output message="tns:space" name="entered"/>
  </operation>
</portType>
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered"/>
  </operation>
</binding>
```

An XML document generated from the input message would be similar to [Example 74](#). Notice that the XML document now only has one root element.

Example 74: XML Document generated using the `rootNode` attribute

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

Overriding the binding's `rootNode` attribute setting

You can also set the `rootNode` attribute for each individual message, or override the global setting for a particular message, by using the `xformat:body` element inside of the message binding. For example, if you wanted the output message defined in [Example 73](#) to have a different root element from the input message, you could override the binding's root element as shown in [Example 75](#).

Example 75: Using `xformat:body`

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered"/>
    <xformat:body rootNode="entryStatus"/>
  </operation>
</binding>
```


Using G2++ Messages

Overview

G2++ is a set of mechanisms for defining and manipulating hierarchically structured messages. G2++ messages can be thought of as records, which are described in terms of their structure and the data types they contain.

G2++ is an alternative to "raw" structures (such as C or C++ structs), which rely on common data representation characteristics that might not be present in a heterogeneous distributed system.

Simple G2++ mapping example

Consider the following instance of a G2++ message:

Note: Because tabs are significant in G2++ files (that is, tabs indicate scoping levels and are not simply treated as "white space"), examples in this chapter indicate tab characters as an up arrow (caret) followed by seven spaces.

Example 76: *ERecord G2++ Message*

```

ERecord
^
^   XYZ_Part
^   ^   XYZ_Code^   someValue1
^   ^   password^   someValue2
^   ^   serviceFieldName^   someValue3
^   newPart
^   ^   newActionCode^   someValue4
^   ^   newServiceClassName^   someValue5
^   ^   oldServiceClassName^   someValue6

```

This G2++ message can be mapped to the following logical description, expressed in WSDL:

Example 77: *WSDL Logical Description of ERecord Message*

```

<types>
  <schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="XYZ_Part">
      <all>
        <element name="XYZ_Code" type="xsd:string"/>
        <element name="password" type="xsd:string"/>
        <element name="serviceFieldName" type="xsd:string"/>
      </all>
    </complexType>
    <complexType name="newPart">
      <all>
        <element name="newActionCode" type="xsd:string"/>
        <element name="newServiceClassName" type="xsd:string"/>
        <element name="oldServiceClassName" type="xsd:string"/>
      </all>
    <complexType name="PRequest">
      <all>
        <element name="newPart" type="xsd1:newPart"/>
        <element name="XYZ_Part" type="xsd1:XYZ_Part"/>
      </all>
    </complexType>

```

Note that each of the message sub-structures (*newPart* and *XYZ_Part*) are initially described separately in terms of their elements, then the two sub-structures are aggregated to form the enclosing record (*PRequest*).

This logical description is mapped to a physical representation of the G2++ message, also expressed in WSDL:

Example 78: WSDL Physical Representation of ERecord Message

```
<binding name="ERecordBinding" type="tns:ERecordRequestPortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <artix:binding transport="tuxedo" format="g2++">
    <G2Definitions>
      <G2MessageDescription name="creation" type="msg">
        <G2MessageComponent name="ERecord" type="struct">
          <G2MessageComponent name="XYZ_Part" type="struct">
            <element name="XYZ_Code" type="element"/>
            <element name="password" type="element"/>
            <element name="serviceFieldName" type="element"/>
          </G2MessageComponent>
          <G2MessageComponent name="newPart" type="struct">
            <element name="newActionCode" type="element"/>
            <element name="newServiceClassName" type="element"/>
            <element name="oldServiceClassName" type="element"/>
          </G2MessageComponent>
        </G2MessageComponent>
      </G2MessageDescription>
    </G2Definitions>
  </artix:binding>
```

Note that all G2++ definitions are contained within the scope of the `G2Definitions` element. Each of the messages are defined within the scope of a `G2MessageDescription` element. The `type` attribute for message descriptions must be `msg` while the `name` attribute simply has to be unique. Each record is described within the scope of a `G2MessageComponent` element. Within this, the `name` attribute must reflect the G2++ record name and the `type` attribute must be `struct`.

Nested within the records are the element definitions; however, if required, a record could be nested here by inclusion of a nested `G2MessageComponent` element (`newPart` and `XYZ_Part` are nested records of parent `ERecord`). Element `name` attributes must match the G2 element name. Defining a record and then referencing it as a nested struct of a parent is legal for the logical mapping but not the physical. In the physical mapping, nested structs must be defined in-place.

The following example illustrates the custom mapping of arrays, which differs from strictly defined G2++ array mappings. The array definition is shown below:

```

IMS_MetaData^      2
^      0
^      ^      columnName^      SERVICENAME
^      ^      columnValue^      someValue1
^      1
^      ^      columnName^      SERVICEACTION
^      ^      columnValue^      someValue2

```

This represents an array with two elements. When placed in a G2++ message, the result is as follows:

Example 79: *Extended ERecord G2++ Message*

```

ERecord
^      XYZ_Part
^      ^      XYZ_Code^      someValue1
^      ^      password^      someValue2
^      ^      serviceFieldName^      someValue3
^      XYZ_MetaData^      1
^      ^      0
^      ^      ^      columnName^      pushToTalk
^      ^      ^      columnValue^      PT01
^      newPart
^      ^      newActionCode^      someValue4
^      ^      newServiceClassName^      someValue5
^      ^      oldServiceClassName^      someValue6

```

In this version of the ERecord record, XYZ_Part contains an array called XYZ_MetaData, whose size is one. The single entry can be thought of as a name/value pair: pushToTalk/PT01, which allows us to ignore columnName and columnValue.

Mapping the new ERecord record to a WSDL logical description results in the following:

Example 80: *WSDL Logical Description of Extended ERecord Message*

```
<types>
  <schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
    <complexType name="XYZ_Part">
      <all>
        <element name="XYZ_Code" type="xsd:string"/>
        <element name="password" type="xsd:string"/>
        <element name="serviceFieldName" type="xsd:string"/>
        <element name="pushToTalk" type="xsd:string"/>
      </all>
    </complexType>
    <complexType name="newPart">
      <all>
        <element name="newActionCode" type="xsd:string"/>
        <element name="newServiceClassName" type="xsd:string"/>
        <element name="oldServiceClassName" type="xsd:string"/>
      </all>
    <complexType name="PRequest">
      <all>
        <element name="newPart" type="xsd1:newPart"/>
        <element name="XYZ_Part" type="xsd1:XYZ_Part"/>
      </all>
    </complexType>
  </schema>
</types>
```

Thus the array elements `columnName` and `columnValue` are “promoted” to a name/value pair in the logical mapping. This physical G2++ representation can now be mapped as follows:

Example 81: *WSDL Physical Representation of Extended ERecord Message*

```
<binding name="ERecordBinding" type="tns:ERecordRequestPortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <artix:binding transport="tuxedo" format="g2++">
    <G2Definitions>
      <G2MessageDescription name="creating" type="msg">
        <G2MessageComponent name="ERecord" type="struct">
          <G2MessageComponent name="XYZ_Part" type="struct">
            <element name="XYZ_Code" type="element"/>
            <element name="password" type="element"/>
            <element name="serviceFieldName" type="element"/>
            <G2MessageComponent name="XYZ_MetaData" type="array" size="1">
              <element name="pushToTalk" type="element"/>
            </G2MessageComponent>
          </G2MessageComponent>
          <G2MessageComponent name="newPart" type="struct">
            <element name="newActionCode" type="element"/>
            <element name="newServiceClassName" type="element"/>
            <element name="oldServiceClassName" type="element"/>
          </G2MessageComponent>
        </G2MessageComponent>
      </G2MessageDescription>
    </G2Definitions>
  </artix:binding>
```

This physical mapping of the extended ERecord message now contains an array, described with its `XYZ_MetaData` name (as per the G2++ record definition). Its type is "array" and its size is one. This `G2MessageComponent` contains a single element called "pushToTalk".

Ignoring unknown elements

It is possible to create a `G2Definitions` element that begins with a G2-specific configuration scope. This configuration scope is called `G2Config` in the following example:

```
<G2Definitions>
^   <G2Config>
^   ^   <IgnoreUnknownElements value="true"/>
</G2Config>
.
.
.
```

In this scope, the only variable used is `IgnoreUnknownElements`, which can have a value of `true` or `false`. If the value is set to `true`, elements or array elements that are not defined in the G2 message definitions will be ignored. For example the following record would be valid if `IgnoreUnknownElements` is set to `true`.

Example 82: Valid G2++ Record With Ignored Fields

```
ERecord
^   XYZ_Part
^   XYZ_Code^   someValue1
^   AnElement^   foo
^   password^   someValue2
^   serviceFieldName^   someValue3
^   XYZ_MetaData^   2
^   ^   0
^   ^   ^   columnName^   pushToTalk
^   ^   ^   columnValue^   PT01
^   ^   1
^   ^   ^   columnName^   AnArrayElement
^   ^   ^   columnValue^   bar
^   newPart
^   ^   newActionCode^   someValue4
^   ^   newServiceClassName^   someValue5
^   ^   oldServiceClassName^   someValue6
```

When parsed, the above `ERecord` would not include the elements "AnElement" or "AnArrayElement". If `IgnoreUnknownElements` is set to `false`, the above record would be rejected as invalid.

Part III

Transports

In this part

This part contains the following chapters:

Understanding How Endpoints are Defined WSDL	page 157
Using HTTP	page 161
Using IIOP	page 181
Using WebSphere MQ	page 187
Using the Java Messaging System	page 203
Using TIBCO Rendezvous	page 217
Using Tuxedo	page 223
Using FTP	page 227

Understanding How Endpoints are Defined WSDL

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

Overview

An endpoint can be thought of as a physical manifestation of a service. It combines a binding, which specifies the physical representation of the logical data used by a service, and a set of networking details that define the physical connection details used to make the service contactable by other endpoints.

Endpoints and services

In the same way a binding can only map a single interface, an endpoint can only map to a single service. However, a service can be manifested by any number of endpoints. For example, you could define a ticket selling service that was manifested by four different endpoints. However, you could not have a single endpoint that manifested both a ticket selling service and a widget selling service.

The WSDL elements

Endpoints are defined in a contract using a combination of the WSDL `service` element and the WSDL `port` element. The `service` element is a collection of related `port` elements. The `port` elements define the actual endpoints.

The WSDL `service` element has a single attribute, `name`, that specifies a unique name. The `service` element is used as the parent element of a collection of related `port` elements. WSDL makes no specification about how the `port` elements are related. You can associate the `port` elements in any manner you see fit.

The WSDL `port` element has a single attribute, `binding`, that specifies the binding used by the endpoint. The `port` element is the parent element of the elements that specify the actual transport details used by the endpoint. The elements used to specify the transport details are discussed in the following sections.

Adding endpoints to a contract

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

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

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different transports used in defining an endpoint work.

You can also add an endpoint to a contract using any text editor. When you hand edit a contract, you are responsible for ensuring that the contract is valid.

Supported transports

Artix endpoint definitions are built using extensions defined for each of the transports Artix supports. Artix supports the following transports:

- HTTP
- BEA Tuxedo
- IBM WebSphere MQ
- TIBCO Rendezvous™
- IIOP

- CORBA
- Java Messaging Service
- File Transfer Protocol

Using HTTP

HTTP is the standard transport used in Web services.

Overview

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

In this chapter

This chapter discusses the following topics:

Adding an HTTP Endpoint to a Contract	page 162
Configuring an HTTP Endpoint	page 169

Adding an HTTP Endpoint to a Contract

Overview

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

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

`soap:address`

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

[Example 83](#) shows a `port` element used to send SOAP over HTTP.

Example 83: *SOAP Port Element*

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

`http:address`

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

Using the command line tool

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

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

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

<code>-transport soap/http</code>	If the payload being sent over the wire is SOAP, use <code>-transport soap</code> . For all other payloads use <code>-transport http</code> .
-----------------------------------	---

<code>-e <i>service</i></code>	Specifies the name of the generated <code>service</code> element.
<code>-t <i>port</i></code>	Specifies the value of the <code>name</code> attribute of the generated <code>port</code> element.
<code>-b <i>binding</i></code>	Specifies the name of the binding for which the service is generated.
<code>-a <i>address</i></code>	Specifies the value used in the <code>address</code> element of the port.
<code>-hssdt <i>serverSendTimeout</i></code>	Specifies the number of milliseconds that the server can continue to try to send a response to the client before the connection is timed-out.
<code>-hscvt <i>serverReceiveTimeout</i></code>	Specifies the number of milliseconds that the server can continue to try to receive a request from the client before the connection is timed-out.
<code>-hstrc <i>trustedRootCertificates</i></code>	Specifies the full path to the X509 certificate for the certificate authority.
<code>-hsuss <i>useSecureSockets</i></code>	Specifies if the server uses secure sockets. Valid values are <code>true</code> or <code>false</code> .
<code>-hsct <i>contentType</i></code>	Specifies the media type of the information being sent in a server response.
<code>-hsc <i>serverCacheControl</i></code>	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.
<code>-hsscse <i>supressClientSendErrors</i></code>	Specifies whether exceptions are thrown when an error is encountered on receiving a client request. Valid values are <code>true</code> or <code>false</code> .
<code>-hsscre <i>supressClientReceiveErrors</i></code>	Specifies whether exceptions are thrown when an error is encountered on sending a response to a client. Valid values are <code>true</code> or <code>false</code> .
<code>-hshka <i>honorKeepAlive</i></code>	Specifies if the server honors client keep-alive requests. Valid values are <code>true</code> or <code>false</code> .
<code>-hsrurl <i>redirectURL</i></code>	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.

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

<code>-hcun</code> <i>userName</i>	Specifies the username the client uses to register with servers.
<code>-hcp</code> <i>password</i>	Specifies the password the client uses to register with servers.
<code>-hcat</code> <i>clientAuthorizationType</i>	Specifies the authorization mechanisms the client uses when contacting servers.
<code>-hca</code> <i>clientAuthorization</i>	Specifies the authorization credentials used to perform the authorization.
<code>-hca</code> <i>accept</i>	Specifies what media types the client is prepared to handle.
<code>-hcal</code> <i>acceptLanguage</i>	Specifies what language the client prefers for the purposes of receiving a response
<code>-hcae</code> <i>acceptEncoding</i>	Specifies what content codings the client is prepared to handle.
<code>-hch</code> <i>host</i>	Specifies the internet host and port number of the resource on which the client request is being invoked.
<code>-hccn</code> <i>clientConnection</i>	Specifies if the client will open a new connection for each request or if it will keep the original one open. Valid values are <code>close</code> and <code>Keep-Alive</code> .
<code>-hcck</code> <i>cookie</i>	Specifies a static cookie to be sent to the server.
<code>-hcbt</code> <i>browserType</i>	Specifies information about the browser from which the client request originates.
<code>-hcr</code> <i>referer</i>	Specifies the value for the client's referring entity.
<code>-hcps</code> <i>proxyServer</i>	Specifies the URL of the proxy server, if one exists along the message path.
<code>-hcpun</code> <i>proxyUserName</i>	Specifies the username that the client uses to be authorized by proxy servers.
<code>-hcpp</code> <i>proxyPassword</i>	Specifies the password that the client uses to be authorized by proxy servers.
<code>-hcpat</code> <i>proxyAuthorizationType</i>	Specifies the authorization mechanism the client uses with proxy servers.
<code>-hcpa</code> <i>proxyAuthorization</i>	Specifies the actual data that the proxy server should use to authenticate the client.

<code>-hccce <i>clientCertificate</i></code>	Specifies the full path to the X509 certificate issued by the certificate authority for the client.
<code>-hcccc <i>clientCertificateChain</i></code>	Specifies the full path to the file that contains all the certificates in the chain.
<code>-hcpk <i>clientPrivateKey</i></code>	Specifies the full path to the private key that corresponds to the X509 certificate specified by <i>clientCertificate</i> .
<code>-hcpkp <i>clientPrivateKeyPassword</i></code>	Specifies a password that is used to decrypt the private key.
<code>-o <i>file</i></code>	Specifies the filename for the generated contract. The default is to append <code>-service</code> to the name of the imported contract.
<code>-d <i>dir</i></code>	Specifies the output directory for the generated contract.
<code>-L <i>file</i></code>	Specifies the location of your Artix license file. The default behavior is to check <code>IT_PRODUCT_DIR\etc\license.txt</code> .
<code>-quiet</code>	Specifies that the tool runs in quiet mode.
<code>-verbose</code>	Specifies that the tool runs in verbose mode.
<code>-h</code>	Displays the tool's usage statement.
<code>-v</code>	Displays the tool's version.

For more information about the specific attributes and their values see the [Artix WSDL Extension Reference](#).

Example

[Example 84](#) shows the namespace entries you need to add to the `definitions` element of your contract to use the HTTP extensions.

Example 84: Artix HTTP Extension Namespaces

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

[Example 85](#) shows a `port` element for an endpoint that sends fixed data over HTTP.

Example 85: *Generic HTTP Port*

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

Configuring an HTTP Endpoint

Overview

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

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

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

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

For a complete discussion of the specific attributes and their values see the [Artix WSDL Extension Reference](#).

In this section

This section discusses the following features:

Specifying Send and Recieve Timeout Limits	page 170
Specifying a Username and a Password	page 172
Configuring Keep-Alive Behavior	page 174
Specifying Cache Control Directives	page 176

Specifying Send and Recieve Timeout Limits

Overview

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

Send Timeout

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

This value may need to be adjusted if you are transmitting large messages as they take longer to send. Other factors that may effect the amount if time needed to transmit messages over HTTP are the speed of the network, distance between the endpoints, and the ammount of traffic on the network. For example, if you were transmitting high-resolution photographs accross the Atlantic, you may need to adjust the value of the `SendTimeout` attribute to 1200000 as shown in [Example 86](#).

Example 86: *Setting the SendTimeout Attribute*

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

Recieve Timeout

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

This value has the same meaning for both client-side and server-side endpoints. It does not specify the ammount of time a client will wait for a response. It only specifies the ammount of time an endpoint spends between when it initially recieves the beginning of a message and the when it recieves the last peice of data in the message. For example, if a client

using the default settings sends a response to a service that takes 90 seconds to process the response, the client will not timeout. However, if it takes the client 45 seconds to read the response from the network, it will timeout.

The causes for long read times are similar to the reasons for long send times. Large messages, heavy network traffic, and large physical distances can all have an impact on the amount of time it takes an HTTP endpoint to receive a message. For example, if you are transmitting map data to a remote research facility, you may want to specify a value of 600000 for the `ReceiveTimeout` attribute of the remote endpoint as shown in [Example 87](#).

Example 87: *Setting the `ReceiveTimeout` Attribute*

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

Specifying a Username and a Password

Overview

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

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

- `UserName`
- `Password`

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

Setting a username

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

Setting a password

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

Relationship between the attributes

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

Similarly, Artix does not require that the two attributes be used together. If an endpoint only needs to provide a password, you can provide a value for the `Password` attribute without providing a value for the `UserName` attribute. [Example 88](#) shows an HTTP endpoint definition that specifies just a username.

Example 88: *Specifying Just a Username*

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

The attributes and other security features

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

Configuring Keep-Alive Behavior

Overview

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

Making keep-alive requests

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

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

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

[Example 89](#) shows a `port` element that defines an HTTP client endpoint that does not want to reuse connections.

Example 89: *Specifying that the HTTP Connection is Closed*

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

Honoring keep-alive requests

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

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

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

[Example 90](#) shows a `port` element that defines an HTTP server endpoint that rejects keep-alive requests.

Example 90: *Rejecting Keep-Alive Requests*

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

Specifying Cache Control Directives

Overview

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

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

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

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

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

Server endpoint settings

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

Table 22 shows the valid values for `CacheControl` in `http-conf:server`.

Table 22: *Settings for CacheControl on an HTTP Server Endpoint*

Directive	Behavior
<code>no-cache</code>	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.
<code>public</code>	Any cache can store the response.
<code>private</code>	Public (<i>shared</i>) 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.
<code>no-store</code>	Caches must not store any part of response or any part of the request that invoked it.
<code>no-transform</code>	Caches must not modify the media type or location of the content in a response between a server and a client.
<code>must-revalidate</code>	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.

Table 22: *Settings for CacheControl on an HTTP Server Endpoint*

Directive	Behavior
proxy-revalidate	Means the same as <code>must-revalidate</code> , except that it can only be enforced on shared caches and is ignored by private unshared caches. If using this directive, the <code>public</code> cache directive must also be used.
max-age	Specifies the maximum age, in seconds, of a cached response before it is stale.
s-maxage	Means the same as <code>max-age</code> , except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by <code>s-maxage</code> overrides the age specified by <code>max-age</code> . If using this directive, the <code>proxy-revalidate</code> directive must also be used.
cache-extension	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

Client endpoint settings

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

[Table 23](#) shows the valid settings for `CacheControl` in `http-conf:client`.

Table 23: *Settings for CacheControl on HTTP Client Endpoint*

Directive	Behavior
<code>no-cache</code>	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.
<code>no-store</code>	Caches must not store any part of a response or any part of the request that invoked it.
<code>max-age</code>	The client can accept a response whose age is no greater than the specified time in seconds.
<code>max-stale</code>	The client can accept a response that has exceeded its expiration time. If a value is assigned to <code>max-stale</code> , 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.
<code>min-fresh</code>	The client wants a response that will be still be fresh for at least the specified number of seconds indicated.
<code>no-transform</code>	Caches must not modify media type or location of the content in a response between a server and a client.

Table 23: *Settings for CacheControl on HTTP Client Endpoint*

Directive	Behavior
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.

Using IIOP

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

Overview

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

Namespace

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

IIOP address specification

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

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

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

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

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

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

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

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

For more information on using the name service with Artix see [Artix for CORBA](#).

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

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

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

Specifying type of payload encoding

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

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

Specifying POA policies

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

- [POA Name](#)
- [Persistence](#)
- [ID Assignment](#)

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

POA Name

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

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

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

Persistence

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

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

ID Assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by Artix. To specify that the IIOP endpoint's POA should use a user-assigned ID, use the following:

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

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

Using the command line tool

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

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

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

- `-e service` Specifies the name of the generated `service` element.
- `-t port` Specifies the value of the `name` attribute of the generated `port` element.
- `-b binding` Specifies the name of the binding for which the endpoint is generated.

<code>-a address</code>	Specifies the value used in the generated <code>iiop:address</code> elements.
<code>-poa poaName</code>	Specifies the value of the POA name policy.
<code>-sid serviceId</code>	Specifies the value of the ID assignment policy.
<code>-pst persists</code>	Specifies the value of the persistence policy. Valid values are <code>true</code> and <code>false</code> .
<code>-paytype payload</code>	Specifies the type of data being sent in the message payloads. Valid values are <code>string</code> , <code>octets</code> , <code>imsraw</code> , <code>imsraw_binary</code> , <code>cicsraw</code> , and <code>cicsraw_binary</code> .
<code>-o file</code>	Specifies the filename for the generated contract. The default is to append <code>-service</code> to the name of the imported contract.
<code>-d dir</code>	Specifies the output directory for the generated contract.
<code>-L file</code>	Specifies the location of your Artix license file. The default behavior is to check <code>IT_PRODUCT_DIR/etc/license.txt</code> .
<code>-quiet</code>	Specifies that the tool runs in quiet mode.
<code>-verbose</code>	Specifies that the tool runs in verbose mode.
<code>-h</code>	Displays the tool's usage statement.
<code>-v</code>	Displays the tool's version.

For more information about the specific attributes and their values see the [Artix WSDL Extension Reference](#).

Example

For example, an IIOP endpoint definition for the `personalInfoLookup` binding would look similar to [Example 91](#):

Example 91: CORBA `personalInfoLookup` Port

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

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

Using WebSphere MQ

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

In this chapter

This chapter discusses the following topics:

Adding a WebSphere MQ Endpoint	page 188
Specifying the WebSphere Library to Load	page 194
Using Queues on Remote Hosts	page 196
Using WebSphere MQ's Transaction Features	page 198
Setting a Value for the Message Descriptor's Format Field	page 200

Adding a WebSphere MQ Endpoint

Overview

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

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

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

WebSphere MQ namespace

The WSDL extensions used to describe WebSphere MQ transport details are defined in the WSDL namespace

`http://schemas.iona.com/transport/mq`. If you are going to add a WebSphere MQ port by hand you will need to include the following in the `definitions` tag of your contract:

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

Required attributes

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

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

Using the command line tool

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

```
wsdltoservice -transport mq [-e service] [-t port] [-b binding]
[-sqm queueManager] [-sqn queue] [-srqm queueManager]
[-srqn queue] [-smqn modelQueue] [-sus usageStyle]
[-scs correlationStyle] [-sam accessMode]
[-sto timeout] [-sme expiry] [-smp priority]
[-smi messageId] [-sci correlationId] [-sd delivery]
[-st transactional] [-sro reportOption] [-sf format]
[-sad applicationData] [-sat accountingToken]
[-scn connectionName] [-sc convert] [-scr reusable]
[-scfp fastPath] [-said idData] [-saod originData]
[-cqmq queueManager] [-cqmq queue] [-crqm queueManager]
[-crqn queue] [-cmqn modelQueue] [-cus usageStyle]
[-ccs correlationStyle] [-cam accessMode]
[-cto timeout] [-cme expiry] [-cmp priority]
[-cmi messageId] [-cci correlationId] [-cd delivery]
[-ct transactional] [-cro reportOption] [-cf format]
[-cad applicationData] [-cat accountingToken]
[-ccn connectionName] [-cc convert] [-ccr reusable]
[-ccfp fastPath] [-caid idData] [-caod originData]
[-caqn queue] [-cui userId] [-o file] [-d dir]
[-L file] [-quiet] [-verbose] [-h] [-v] wsdurl
```

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

<code>-e service</code>	Specifies the name of the generated <code>service</code> element.
<code>-t port</code>	Specifies the value of the <code>name</code> attribute of the generated <code>port</code> element.
<code>-b binding</code>	Specifies the name of the binding for which the endpoint is generated.
<code>-sqm queueManager</code>	Specifies the name of the server's queue manager.
<code>-sqn queue</code>	Specifies the name of the server's request queue.
<code>-srqm queueManager</code>	Specifies the name of the server's reply queue manager.
<code>-srqn queue</code>	Specifies the name of the server's reply queue.
<code>-smqn modelQueue</code>	Specifies the name of the server's model queue.

<code>-sus usageStyle</code>	Specifies the value of the server's <code>UsageStyle</code> attribute. Valid values are <code>Peer</code> , <code>Requester</code> , or <code>Responder</code> .
<code>-scs correlationStyle</code>	Specifies the value of the server's <code>CorrelationStyle</code> attribute. Valid values are <code>messageId</code> , <code>correlationId</code> , or <code>messageId copy</code> .
<code>-sam accessMode</code>	Specifies the value of the server's <code>AccessMode</code> attribute. Valid values are <code>peek</code> , <code>send</code> , <code>receive</code> , <code>receive exclusive</code> , or <code>receive shared</code> .
<code>-sto timeout</code>	Specifies the value of the server's <code>Timeout</code> attribute.
<code>-sme expiry</code>	Specifies the value of the server's <code>MessageExpiry</code> attribute.
<code>-smp priority</code>	Specifies the value of the server's <code>MessagePriority</code> attribute.
<code>-smi messageId</code>	Specifies the value of the server's <code>MessageId</code> attribute.
<code>-sci correlationId</code>	Specifies the value of the server's <code>CorrelationID</code> attribute.
<code>-sd delivery</code>	Specifies the value of the server's <code>Delivery</code> attribute.
<code>-st transactional</code>	Specifies the value of the server's <code>Transactional</code> attribute. Valid values are <code>none</code> , <code>internal</code> , or <code>xa</code> .
<code>-sro reportOption</code>	Specifies the value of the server's <code>ReportOption</code> attribute. Valid values are <code>none</code> , <code>coa</code> , <code>cod</code> , <code>exception</code> , <code>expiration</code> , or <code>discard</code> .
<code>-sf format</code>	Specifies the value of the server's <code>Format</code> attribute.
<code>-sad applicationData</code>	Specifies the value of the server's <code>ApplicationData</code> attribute.
<code>-sat accountingToken</code>	Specifies the value of the server's <code>AccountingToken</code> attribute.
<code>-scn connectionName</code>	Specifies the name of the connection by which the adapter connects to the queue.
<code>-sc convert</code>	Specifies if the messages in the queue need to be converted to the system's native encoding. Valid values are <code>true</code> or <code>false</code> .

<code>-scr reusable</code>	Specifies the value of the server's <code>ConnectionReusable</code> attribute. Valid values are true or false.
<code>-scfp fastPath</code>	Specifies the value of the server's <code>ConnectionFastPath</code> attribute. Valid values are true or false.
<code>-said idData</code>	Specifies the value of the server's <code>ApplicationIdData</code> attribute.
<code>-saod originData</code>	Specifies the value of the server's <code>ApplicationOriginData</code> attribute.
<code>-cqm queueManager</code>	Specifies the name of the client's queue manager.
<code>-cqn queue</code>	Specifies the name of the client's request queue.
<code>-crqm queueManager</code>	Specifies the name of the client's reply queue manager.
<code>-crqn queue</code>	Specifies the name of the client's reply queue.
<code>-cmqn modelQueue</code>	Specifies the name of the client's model queue.
<code>-cus usageStyle</code>	Specifies the value of the client's <code>UsageStyle</code> attribute. Valid values are <code>Peer</code> , <code>Requester</code> , or <code>Responder</code> .
<code>-ccs correlationStyle</code>	Specifies the value of the client's <code>CorrelationStyle</code> attribute. Valid values are <code>messageId</code> , <code>correlationId</code> , or <code>messageId copy</code> .
<code>-cam accessMode</code>	Specifies the value of the client's <code>AccessMode</code> attribute. Valid values are <code>peek</code> , <code>send</code> , <code>receive</code> , <code>receive exclusive</code> , or <code>receive shared</code> .
<code>-cto timeout</code>	Specifies the value of the client's <code>Timeout</code> attribute.
<code>-cme expiry</code>	Specifies the value of the client's <code>MessageExpiry</code> attribute.
<code>-cmp priority</code>	Specifies the value of the client's <code>MessagePriority</code> attribute.
<code>-cmi messageId</code>	Specifies the value of the client's <code>MessageId</code> attribute.
<code>-cci correlationId</code>	Specifies the value of the client's <code>CorrelationId</code> attribute.

<code>-cd <i>delivery</i></code>	Specifies the value of the client's <code>Delivery</code> attribute.
<code>-ct <i>transactional</i></code>	Specifies the value of the client's <code>Transactional</code> attribute. Valid values are <code>none</code> , <code>internal</code> , or <code>xa</code> .
<code>-cro <i>reportOption</i></code>	Specifies the value of the client's <code>ReportOption</code> attribute. Valid values are <code>none</code> , <code>coa</code> , <code>cod</code> , <code>exception</code> , <code>expiration</code> , or <code>discard</code> .
<code>-cf <i>format</i></code>	Specifies the value of the client's <code>Format</code> attribute.
<code>-cad <i>applicationData</i></code>	Specifies the value of the client's <code>ApplicationData</code> attribute.
<code>-cat <i>accountingToken</i></code>	Specifies the value of the client's <code>AccountingToken</code> attribute.
<code>-ccn <i>connectionName</i></code>	Specifies the name of the connection by which the adapter connects to the queue.
<code>-cc <i>convert</i></code>	Specifies if the messages in the queue need to be converted to the system's native encoding. Valid values are <code>true</code> or <code>false</code> .
<code>-ccr <i>reusable</i></code>	Specifies the value of the client's <code>ConnectionReusable</code> attribute. Valid values are <code>true</code> or <code>false</code> .
<code>-ccfp <i>fastPath</i></code>	Specifies the value of the client's <code>ConnectionFastPath</code> attribute. Valid values are <code>true</code> or <code>false</code> .
<code>-caid <i>idData</i></code>	Specifies the value of the client's <code>ApplicationIdData</code> attribute.
<code>-caod <i>originData</i></code>	Specifies the value of the client's <code>ApplicationOriginData</code> attribute.
<code>-caqn <i>queue</i></code>	Specifies the remote queue to which a server will put replies if its queue manager is not on the same host as the client's local queue manager.
<code>-cui <i>userId</i></code>	Specifies the value of the client's <code>UserIdentification</code> attribute.
<code>-o <i>file</i></code>	Specifies the filename for the generated contract. The default is to append <code>-service</code> to the name of the imported contract.
<code>-d <i>dir</i></code>	Specifies the output directory for the generated contract.

-L <i>file</i>	Specifies the location of your Artix license file. The default behavior is to check IT_PRODUCT_DIR\etc\license.txt.
-quiet	Specifies that the tool runs in quiet mode.
-verbose	Specifies that the tool runs in verbose mode.
-h	Displays the tool's usage statement.
-v	Displays the tool's version.

For more information about the specific attributes and their values see the [Artix WSDL Extension Reference](#).

Example

An Artix contract exposing an interface, `monsterBash`, bound to a SOAP payload format, `Raydon`, on an WebSphere MQ queue, `UltraMan` would contain a `service` element similar to [Example 92](#).

Example 92: Sample WebSphere MQ Port

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

Specifying the WebSphere Library to Load

Overview

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

The attribute

Both the `mq:server` element and the `mq:client` element support the attribute that is used to specify which MQ libraries to load. The `Server_Client` attribute specifies which shared libraries to load on systems with a full WebSphere MQ installation. [Table 24](#) describes the settings for this attribute for each type of WebSphere MQ installation.

Table 24: *WebSphere MQ Server_Client Attribute Settings*

MQ Installation	Server_Client Setting	Behavior
Full		The server shared library(<code>libmqm</code>) is loaded and the application will use queues hosted on the local machine.
Full	<code>server</code>	The server shared library(<code>libmqm</code>) is loaded and the application will use queues hosted on the local machine.
Full	<code>client</code>	The client shared library(<code>libmqic</code>) is loaded and the application will use queues hosted on a remote machine.
Client		The application will attempt to load the server shared library(<code>libmqm</code>) before loading the client shared library(<code>libmqic</code>). The application accesses queues hosted on a remote machine.

Table 24: *WebSphere MQ Server_Client Attribute Settings*

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

Example

[Example 93](#) shows an a `service` element for an MQ endpoint that uses the MQ client shared library.

Example 93: *ARTIX MQ Endpoint Using MQ Client Library*

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

Using Queues on Remote Hosts

Overview

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

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

Effect of AliasQueueName

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

Example

If you had a system defined similar to that shown in [Figure 1](#), you would need to use the `AliasQueueName` attribute setting when configuring your WebSphere MQ client. In this set up the client is running on a host with a local queue manager `QMGR.A`. `QMGR.A` has two queues configured. `RQA` is a remote queue that is a proxy for `RQB` and `RPLA` is a local queue. The server is running on a different machine whose local queue manager is `QMGR.B`.

QM_{grB} also has two queues. R_{qB} is a local queue and R_{plyB} is a remote queue that is a proxy for R_{plyA}. The client places its request on R_{qA} and expects replies to arrive on R_{plyA}.

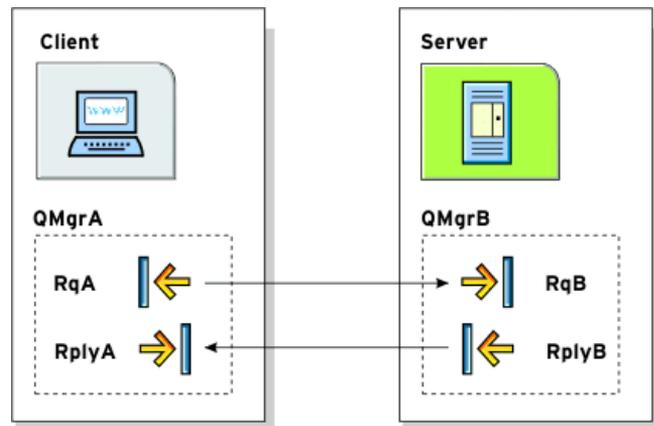


Figure 1: MQ Remote Queues

The `port` elements for the client and server for this deployment are shown in [Example 94](#). The `AliasQueueName` attribute is set to `RplyB` because that is the remote queue proxying for the reply queue in server's local queue manager. The `ReplyQueueManager` attribute and the `ReplyQueueName` attribute are set to the client's local queue manager so that it knows where to listen for responses. In this example, the server's `ReplyQueueManager` attribute and `ReplyQueueName` attribute do not need to be set because you are assured that the client is populating the request's message descriptor with the needed information for the server to determine where replies are sent.

Example 94: Setting Up WebSphere MQ Ports for Intercommunication

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

Using WebSphere MQ's Transaction Features

Overview

Artix endpoints that use WebSphere MQ as their transport can take advantage of WebSphere MQ's internal transaction features. By using these features you can make WebSphere MQ a reliable message transport.

WebSphere MQ's transaction features can also interact with Artix's transaction features. For a complete description of using Artix's transaction features and how they interact with WebSphere MQ, see either [Artix Transactions Guide, C++](#) or [Artix Transactions Guide, Java](#).

Specifying transaction style

You specify how an Artix endpoint uses WebSphere MQ's transaction features using the `Transactional` attribute. Both the `mq:client` element and the `mq:server` element support the `Transactional` attribute.

The values of the `Transactional` attribute are explained in [Table 25](#).

Table 25: *WebSphere MQ Transactional Attribute Settings*

Attribute Setting	Description
<code>none</code> (Default)	The messages are not part of a transaction. No rollback actions will be taken if errors occur.
<code>internal</code>	The messages involved in an invocation are part of a transaction with WebSphere MQ serving as the transaction manager. Each invocation is a separate transaction.
<code>xa</code>	The messages involved in an invocation are part of a flowed transaction with WebSphere MQ serving as an enlisted resource manager. Each invocation is a separate transaction.

Correlation to persistence

If you set your MQ endpoint to use one of the transactional styles, you must also ensure that it uses persistent messages. You do this by setting the `Delivery` attribute of the element defining the endpoint to `persistent`. For

example if you are defining an endpoint that represents an MQ server that uses internal transactions, you set the `mq:server` element's `Delivery` attribute to `persistent`.

Reliable MQ messages

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

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

Example

[Example 95](#) shows the `mq:server` element for an MQ endpoint whose requests will be part of transactions managed by WebSphere MQ. Note that the `Delivery` attribute must be set to `persistent` when using transactions.

Example 95: MQ Client Setup to use Transactions

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

Setting a Value fo the Message Descriptor's Format Field

Overview

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

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

Special values

The `Format` attribute can take the special values `none`, `string`, `event`, `programmable command`, and `unicode`. These settings are described in [Table 26](#).

Table 26: *WebSphere MQ Format Attribute Settings*

Attribute Setting	Description
<code>none</code> (Default)	Corresponds to <code>MQFMT_NONE</code> . No format name is specified.
<code>string</code>	Corresponds to <code>MQFMT_STRING</code> . <code>string</code> specifies that the message consists entirely of character data. The message data may be either single-byte characters or double-byte characters.
<code>unicode</code>	Corresponds to <code>MQFMT_STRING</code> . <code>unicode</code> specifies that the message consists entirely of Unicode characters. (Unicode is not supported in Artix at this time.)

Table 26: *WebSphere MQ Format Attribute Settings (Continued)*

Attribute Setting	Description
event	Corresponds to MQFMT_EVENT. event specifies that the message reports the occurrence of an WebSphere MQ event. Event messages have the same structure as programmable commands.
programmable command	Corresponds to MQFMT_PCF. programmable command specifies that the messages are user-defined messages that conform to the structure of a programmable command format (PCF) message. For more information, consult the IBM Programmable Command Formats and Administration Interfaces documentation at http://publibfp.boulder.ibm.com/epubs/html/csqa03/csqa030d.htm#Header_12 .

Using Codeset Conversion

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

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

Example

[Example 96](#) shows an `mq:client` element that defines an endpoint used for making requests against a server on a mainframe system.

Example 96: *WebSphere MQ Client Talking to the Mainframe*

```
<mq:client QueueManager="hunter" QueueName="bigGuy"  
  ReplyQueueManager="slate" ReplyQueueName="rusty"  
  Format="string" Convert="true"/>
```

Using the Java Messaging System

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

In this chapter

This chapter discusses the following topics:

Defining a JMS Endpoint	page 204
Migrating to the 4.x JMS WSDL Extensions	page 215
Using ActiveMQ as Your JMS Provider	page 216

Defining a JMS Endpoint

Overview

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

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

Message formatting

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

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

Namespace

The WSDL extensions used to define a JMS endpoint are specified in the namespace `http://celtix.objectweb.org/transport/jms`. To use the JMS extensions you will need to add the line shown [Example 97](#) to the `definitions` element of your contract.

Example 97: JMS Extension's Namespace

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

In this section

This section discusses the following topics:

Basic Endpoint Configuration	page 206
Client Endpoint Configuration	page 210
Server Endpoint Configuration	page 211
Using the Command Line Tool	page 213

Basic Endpoint Configuration

Overview

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

address element

The basic configuration for a JMS endpoint is done by using a `jms:address` element in your service's `port` element. The `jms:address` element uses the attributes described in [Table 27](#) to configure the connection to the JMS broker.

Table 27: *JMS Port Attributes*

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

JMSNamingProperties element

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

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

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

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

Using a named reply destination

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

Examples

[Example 98](#) shows an example of an Artix JMS port specification that uses dynamic queues.

Example 98: Artix JMS Port with Dynamic Queues

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

[Example 98](#) shows an example of an Artix JMS port specification that does not use dynamic queues.

Example 99: Artix JMS Port with Non-dynamic Queues

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

Alternate InitialContextFactory settings for using SonicMQ

If you are using Sonic MQ, you will need to use an alternative method of specifying the `InitialContextFactory` value. You specify a colon-separated list of package prefixes to force the JNDI service to instantiate a context

factory with the class name `com.iona.jbus.jms.naming.sonic.sonicURLContextFactory` to perform lookups. This is shown in [Example 100](#).

Example 100: *JMS Port with Alternate InitialContextFactory Specification*

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

Using the contract in [Example 100](#), Artix would use the URL `sonic:jms/queue/helloWorldQueue` to get a reference to the desired queue. Artix would be handed a reference to a queue named `helloWorldQueue` if the JMS broker has such a queue.

Client Endpoint Configuration

Overview

JMS client endpoints can be configured to use different types of messages.

client element

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

<code>messageType</code>	Specifies how the message data will be packaged as a JMS message. <code>text</code> specifies that the data will be packaged as a <code>TextMessage</code> . <code>binary</code> specifies that the data will be packaged as an <code>ObjectMessage</code> .
--------------------------	--

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

Server Endpoint Configuration

Overview

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

server element

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

<code>useMessageIDAsCorrealationID</code>	Specifies whether JMS will use the message ID to correlate messages. The default is <code>false</code> .
<code>durableSubscriberName</code>	Specifies the name used to register a durable subscription. See “Setting up durable subscriptions” on page 211
<code>messageSelector</code>	Specifies the string value of a message selector to use. See “Using message selectors” on page 211 .
<code>transactional</code>	Specifies whether the local JMS broker will create transactions around message processing. The default is <code>false</code> . See “Using reliable messaging” on page 212 .

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

Setting up durable subscriptions

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

Using message selectors

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

Using reliable messaging

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

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

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

Using the Command Line Tool

Overview

The `wsdltoservice` tool can add a JMS endpoint definition to your contract.

`wsdltoservice`

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

```
wsdltoservice -transport jms [-e service] [-t port]
                [-b binding] [-o file] [-d dir]
                [-jnp propName:propVal]* [-jds (queue/topic)]
                [-jnf connectionFactoryName]
                [-jdn destinationName]
                [-jrdsn replyDesinationName]
                [-jcun username] [-jcp password]
                [-jmt (text/binary)] [-jms messageSelector]
                [-jumi (true/false)] [-jtr (true/false)]
                [-jdsn durableSubscriber]
                [-L file] [-quiet] [-verbose] [-h] [-v] wSDLurl
```

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

<code>-e service</code>	Specifies the name of the generated service element.
<code>-t port</code>	Specifies the value of the <code>name</code> attribute of the generated port element.
<code>-b binding</code>	Specifies the name of the binding for which the service is generated.
<code>-o file</code>	Specifies the filename for the generated contract. The default is to append <code>-service</code> to the name of the imported contract.
<code>-d dir</code>	Specifies the output directory for the generated contract.
<code>-jnp propName:propVal</code>	Specifies any optional Java properties to use in connecting to the JNDI provider. This information is used to populate a <code>JMSNamingProperty</code> element. You can use this flag multiple times.

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

For more information about the specific attributes and their values see the [Artix WSDL Extension Reference](#).

Migrating to the 4.x JMS WSDL Extensions

Overview

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

XSLT script

The XSLT script used to migrate old JMS endpoint definitions to 4.x JMS endpoints is called `oldjmswsdl_to_newjmswsdl.xsl` and it is located in `InstallDir/Artix/Version/etc/xslt/utilities/jms`. It will take any Artix contract containing a pre-4.x Artix JMS endpoint definition as input and output an equivalent Artix contract containing a 4.x Artix JMS endpoint.

Using the script with Artix

You can use Artix's XSLT processor to convert your JMS endpoints. To do so you run the Artix `xslttransform` command line tool using the options shown in [Example 101](#).

Example 101: *Running the Transformer with the JMS Migration Script*

```
xslttransform -XSL oldjmswsdl_to_newjmswsdl.xsl -IN oldWsd1.wsdl -OUT newWsd1.wsdl
```

The XSLT processor will read the contract in `oldWsd1.wsdl`, transform the old JMS endpoint to a new JMS endpoint, and save the resulting contract in `newWsd1.wsdl`.

Using ActiveMQ as Your JMS Provider

Overview

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

Setting the CLASSPATH

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

If you do not want to set the Artix environment before starting ActiveMQ you need to add

`InstallDir/lib/activemq/activemq/3.2.1/activemq-rt.jar` to your `CLASSPATH`.

Starting the broker

To start the ActiveMQ JMS broker run the following command:

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

Stopping the broker

To shutdown the ActiveMQ JMS broker run the following command:

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

Security

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

More information

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

Using TIBCO Rendezvous

TIBCO Rendezvous is used in a number of enterprise settings.

Overview

The TIBCO Rendezvous transport lets you use Artix to integrate systems based on TIBCO Rendezvous (TIB/RV) software.

Supported Features

[Table 28](#) shows the matrix of TIBCO Rendezvous features Artix supports.

Table 28: *Supported TIBCO Rendezvous Features*

Feature	Supported	Not Supported
Server-Side Advisory Callbacks	x	
Certified Message Delivery	x	
Fault Tolerance (TibrvFtMember/Monitor)		x
Virtual Connections (TibrvVcTransport)		x
Secure Daemon (rvsd/TibrvSDContext)		x
TIBRVMSG_IPADDR32		x
TIBRVMSG_IPPORT16		x

Namespace

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

```
xmlns:tibrv="http://schemas.iona.com/transport/tibrv"
```

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

Describing the port

As with other transports, the TIB/RV transport specifications are contained within a `port` element. Artix uses `tibrv:port` to describe the attributes of a TIB/RV endpoint. The only required attribute for a `tibrv:port` element is `serverSubject` which specifies the subject to which the server listens.

Using the command line tools

To use `wsdltoservice` to add a TIB/RV endpoint use the following options.

```
wsdltoservice -transport tibrv [-e service] [-t port] [-b binding]
               [-tss subject] [-tcst subject] [-tbt bindingType]
               [-tcl callbackLevel] [-trdt timeout]
               [-tts transportService] [-ttn transportNetwork]
               [-ttbm batchMode] [-tqp priority]
               [-tqlp queueLimitPolicy] [-tqme queueMaxEvents]
               [-tqda queueDiscardAmount] [-tcs cmSupport]
               [-tctsn cmTransportServerName]
               [-tctcn cmTransportClientName]
               [-tctro cmTransportRequestOld]
               [-tctl n cmTransportLedgerName]
               [-tctsl cmTransportSyncLedger]
               [-tctra cmTransportRelayAgent]
               [-tctdtl cmTransportDefaultTimeLimit]
```

```

[-tclca cmListenerCancelAgreements]
[-tcqtsn cmQueueTransportServerName]
[-tcqctcn cmQueueTransportClientName]
[-tcqctww cmQueueTransportWorkerWeight]
[-tcqctws cmQueueTransportWorkerTasks]
[-tcqctsw cmQueueTransportSchedulerWeight]
[-tcqctsh cmQueueTransportSchedulerHeartbeat]
[-tcqctsa cmQueueTransportSchedulerActivation]
[-tcqctct cmQueueTransportCompleteTime]
[-tmnfv messageNameFieldValue]
[-tmnfp messageNameFieldPath]
[-tbfi bindingFieldId] [-tbfn bindingFieldName]
[-o file] [-d dir] [-L file]
[-quiet] [-verbose] [-h] [-v] wsdurl

```

The `-transport tibrv` flag specifies that the tool is to generate a TIB/RV service. The other options are as follows.

<code>-e service</code>	Specifies the name of the generated service element.
<code>-t port</code>	Specifies the value of the <code>name</code> attribute of the generated <code>port</code> element.
<code>-b binding</code>	Specifies the name of the binding for which the endpoint is generated.
<code>-tss subject</code>	Specifies the subject to which the server listens.
<code>-tcst subject</code>	Specifies the the prefix to the subject on which the client listens for replies.
<code>-tbt bindingType</code>	Specifies the message binding type. Valid values are <code>msg</code> , <code>xml</code> , <code>opaque</code> , or <code>string</code> .
<code>-tcl callbackLevel</code>	Specifies the server-side callback level when TIB/RV system advisory messages are received. Valid values are <code>INFO</code> , <code>WARN</code> , or <code>ERROR</code> .
<code>-trdt timeout</code>	Specifies the client-side response receive dispatch timeout.
<code>-tts transportService</code>	Specifies the UDP service name or port for <code>TibrvNetTransport</code> .
<code>-ttn transportNetwork</code>	Specifies the binding network addresses for <code>TibrvNetTransport</code> .

<code>-ttbm <i>batchMode</i></code>	Specifies if the TIB/RV transport uses batch mode to send messages. Valid values are <code>DEFAULT_BATCH</code> and <code>TIMER_BATCH</code> .
<code>-tqp <i>priority</i></code>	Specifies the queue priority.
<code>-tqlp <i>queueLimitPolicy</i></code>	Valid values are <code>DISCARD_NONE</code> , <code>DISCARD_NEW</code> , <code>DISCARD_FIRST</code> , or <code>DISCARD_LAST</code> .
<code>-tqme <i>queueMaxEvents</i></code>	Specifies the queue max events.
<code>-tqda <i>queueDiscardAmount</i></code>	Specifies the queue discard amount.
<code>-tcs <i>cmSupport</i></code>	Specifies if Certified Message Delivery support is enabled. Valid values are <code>true</code> or <code>false</code> .
<code>-tctsn <i>cmTransportServerName</i></code>	Specifies the server's <code>TibrvCmTransport</code> correspondent name.
<code>-tctcn <i>cmTransportClientName</i></code>	Specifies the client <code>TibrvCmTransport</code> correspondent name.
<code>-tctro <i>cmTransportRequestOld</i></code>	Specifies if the endpoint can request old messages on start-up. Valid values are <code>true</code> or <code>false</code> .
<code>-tctlm <i>cmTransportLedgerName</i></code>	Specifies the <code>TibrvCmTransport</code> ledger file.
<code>-tctsl <i>cmTransportSyncLedger</i></code>	Specifies if the endpoint uses a synchronous ledger. Valid values are <code>true</code> or <code>false</code> .
<code>-tctra <i>cmTransportRelayAgent</i></code>	Specifies the endpoint's <code>TibrvCmTransport</code> relay agent.
<code>-tctdtl</code> <code><i>cmTransportDefaultTimeLimit</i></code>	Specifies the default time limit for a Certified Message to be delivered.
<code>-tclca</code> <code><i>cmListenerCancelAgreements</i></code>	Specifies if Certified Message agreements are canceled when the endpoint disconnects. Valid values are <code>true</code> or <code>false</code> .
<code>-tcqtsn</code> <code><i>cmQueueTransportServerName</i></code>	Specifies the server's <code>TibrvCmQueueTransport</code> correspondent name.

<code>-tcqtcn</code>	Specifies the client's
<code>cmQueueTransportClientName</code>	TibrvCmQueueTransport correspondent name.
<code>-tcqtww</code>	Specifies the endpoint's
<code>cmQueueTransportWorkerWeight</code>	TibrvCmQueueTransport worker weight.
<code>-tcqtws</code>	Specifies the endpoint's
<code>cmQueueTransportWorkerTasks</code>	TibrvCmQueueTransport worker tasks parameter.
<code>-tcqtsw</code>	Specifies the TibrvCmQueueTransport
<code>cmQueueTransportSchedulerWeight</code>	scheduler weight parameter.
<code>-tcqtsh</code>	Specifies the endpoint's
<code>cmQueueTransportSchedulerHeartbeat</code>	TibrvCmQueueTransport scheduler heartbeat parameter.
<code>-tcqtssa</code>	Specifies the TibrvCmQueueTransport
<code>cmQueueTransportSchedulerActivation</code>	scheduler activation parameter.
<code>-tcqtct</code>	Specifies the TibrvCmQueueTransport
<code>cmQueueTransportCompleteTime</code>	complete time parameter.
<code>-tmfv messageNameFieldValue</code>	Specifies the message name field value.
<code>-tmfp messageNameFieldPath</code>	Specifies the message name field path.
<code>-tbfi bindingFieldId</code>	Specifies the binding field id.
<code>-tbfn bindingFieldName</code>	Specifies the binding field name.
<code>-o file</code>	Specifies the filename for the generated contract. The default is to append <code>-service</code> to the name of the imported contract.
<code>-d dir</code>	Specifies the output directory for the generated contract.
<code>-L file</code>	Specifies the location of your Artix license file. The default behavior is to check <code>IT_PRODUCT_DIR\etc\license.txt</code> .
<code>-quiet</code>	Specifies that the tool runs in quiet mode.
<code>-verbose</code>	Specifies that the tool runs in verbose mode.
<code>-h</code>	Displays the tool's usage statement.
<code>-v</code>	Displays the tool's version.

For more information about the specific attributes and their values see the [Artix WSDL Extension Reference](#).

Example

[Example 102](#) shows an Artix description for a TIB/RV endpoint.

Example 102:*TIB/RV Port Description*

```
<service name="BaseService">
  <port binding="tns:BasePortBinding" name="BasePort">
    <tibrv:port serverSubject="Artix.BaseService.BasePort"/>
  </port>
</service>
```

Using Tuxedo

Overview

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

Tuxedo namespaces

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

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

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

Defining the Tuxedo services

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

Mapping operations to a Tuxedo service

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

Using the command line tools

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

```
wsdltoservice -transport tuxedo [-e service] [-t port]
                [-b binding] [-tsn tuxService]
                [-tfn tuxService:tuxFunction]
                [-ton tuxService:operation]
                [-o file] [-d dir] [-L file] [-quiet] [-verbose] [-h] [-v]
                wSDLurl
```

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

<code>-e <i>service</i></code>	Specifies the name of the generated service element.
<code>-t <i>port</i></code>	Specifies the value of the <code>name</code> attribute of the generated <code>port</code> element.
<code>-b <i>binding</i></code>	Specifies the name of the binding for which the endpoint is generated.
<code>-tsn <i>tuxService</i></code>	Specifies the name of the Tuxedo bulletin board to which Artix connects.
<code>-tfn <i>tuxService:tuxFunction</i></code>	Specifies the name of the function to be used on the specified Tuxedo bulletin board.
<code>-ton <i>tuxService:operation</i></code>	Specifies the WSDL operation that is handled by the specified Tuxedo endpoint.
<code>-o <i>file</i></code>	Specifies the filename for the generated contract. The default is to append <code>-service</code> to the name of the imported contract.
<code>-d <i>dir</i></code>	Specifies the output directory for the generated contract.
<code>-L <i>file</i></code>	Specifies the location of your Artix license file. The default behavior is to check <code>IT_PRODUCT_DIR\etc\license.txt</code> .
<code>-quiet</code>	Specifies that the tool runs in quiet mode.
<code>-verbose</code>	Specifies that the tool runs in verbose mode.

-h Displays the tool's usage statement.
-v Displays the tool's version.

For more information about the specific attributes and their values see the [Artix WSDL Extension Reference](#).

Example

An Artix contract exposing the `personalInfoService` as a Tuxedo endpoint would contain a `service` element similar to [Example 103 on page 225](#).

Example 103: *Tuxedo Port Description*

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


Using FTP

Overview

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

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

In this chapter

This chapter discusses the following topics:

Adding an FTP Endpoint	page 228
Coordinating Requests and Responses	page 230

Adding an FTP Endpoint

Overview

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

FTP namespace

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

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

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

Defining the connection details

The connection details for the endpoint are defined in an `ftp:port` element. The `ftp:port` element has two attributes: `host` and `port`.

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

[Example 104](#) shows an example of a `port` element defining an FTP endpoint.

Example 104: Defining an FTP Endpoint

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

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

`requestLocation` Specifies the location on the FTP server where requests are stored. The default is `/`.

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

Specifying optional naming properties

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

The default naming implementation provided with Artix has two properties:

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

For information on defining optional properties see [“Using Properties to Control Coordination Behavior” on page 239](#).

Coordinating Requests and Responses

Overview

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

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

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

Default implementation

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

In this section

This section discusses the following topics:

Implementing the Client's Coordination Logic	page 231
Implementing the Server's Coordination Logic	page 235
Using Properties to Control Coordination Behavior	page 239

Implementing the Client's Coordination Logic

Overview

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

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

The filename factory

The client-side filename factory is created by implementing the interface `com.iona.jbus.transports.ftp.policy.client.FilenameFactory`. [Example 105](#) shows the interface.

Example 105: Client-Side Filename Factory Interface

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

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

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

    String getNextRequestFilename() throws Exception;

    String getRequestIncompleteFilename(String requestFilename)
    throws Exception;

    String getReplyFilename(String requestFilename)
    throws Exception;

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

The interface has four methods to implement:

initialize()

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

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

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

1. Determine if the user wants to use static filenames based on an `ftp:property` element in the contract. For more information see [“Using Properties to Control Coordination Behavior” on page 239](#).
2. If so, it generates a static filename prefix for the requests.
3. If not, it uses the user supplied filename prefix for the requests.

getNextRequestFilename()

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

getRequestIncompleteFilename()

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

getReplyFilename()

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

The reply lifecycle policy

The reply lifecycle policy is created by implementing the `com.iona.jbus.transports.ftp.policy.client.ReplyFileLifecycle` interface. [Example 106](#) shows the interface.

Example 106: Reply Lifecycle Interface

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

public interface ReplyFileLifecycle
{
    boolean shouldDeleteReplyFile(String fileName)
        throws Exception;

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

The interface has two methods to implement:

shouldDeleteReplyFile()

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

renameReplyFile()

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

Configuring the client's coordination logic

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

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

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

[Example 107](#) shows an example of an Artix configuration scope that specifies an FTP client endpoint's coordination policies.

Example 107: *Configuring an FTP Client Endpoint*

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

For more information on configuring Artix see [Configuring and Deploying Artix Solutions](#).

Implementing the Server's Coordination Logic

Overview

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

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

The filename factory

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

[Example 108](#) shows the interface.

Example 108: *Server-Side Filename Factory Interface*

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

import javax.xml.namespace.QName;

import com.iona.jbus.Bus;
import com.iona.transports.ftp.Element;
import com.iona.webservices.wsdl.ext.ftp.FTPProperties;

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

    String getRequestFileNamesRegEx() throws Exception;

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

    String getReplyIncompleteFilename(String requestFilename)
    throws Exception;

    String getReplyFilename(String requestFilename)
    throws Exception;

    FilenameFactoryPropertyMetaData[] getPropertiesMetaData();
}
```

The interface has six methods to implement:

initialize()

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

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

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

1. Determine if the user wants to use static filenames based on an `ftp:property` element in the contract. For more information see [“Using Properties to Control Coordination Behavior” on page 239](#).
2. If so, it generates a static filename prefix for the requests.
3. If not, it uses the user supplied filename prefix for the requests.

getRequestFileRegEx()

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

updateRequestFiles()

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

getReplyIncompleteFilename()

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

getReplyFilename()

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

getPropertiesMetaData()

`getPropertiesMetaData()` is a convenience function that returns an array of all the possible properties you can use to effect the behavior of the FTP naming scheme. The properties returned correspond to the values defined in the `ftp:properties` element. For more information see [“Using Properties to Control Coordination Behavior” on page 239](#).

The request lifecycle policy

The request lifecycle policy is created by implementing the `com.iona.jbus.transports.ftp.policy.server.RequestFileLifecycle` interface. [Example 109](#) shows the interface.

Example 109: Request Lifecycle Interface

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

public interface RequestFileLifecycle
{
    boolean shouldDeleteRequestFile(String fileName)
        throws Exception;

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

The interface has two methods to implement:

shouldDeleteRequestFile()

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

renameRequestFile()

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

Configuring the server's coordination logic

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

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

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

[Example 110](#) shows an example of an Artix configuration scope that specifies an FTP server endpoint's coordination policies.

Example 110:Configuring an FTP Server Endpoint

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

For more information on configuring Artix see [Configuring and Deploying Artix Solutions](#).

Using Properties to Control Coordination Behavior

Overview

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

Properties in the contract

You can place any number of custom properties into port element defining an FTP endpoint. As described in [“Specifying optional naming properties” on page 229](#), the `ftp:properties` element is a container for one or more `ftp:property` elements. The `ftp:property` element has two attributes: `name` and `value`. Both attributes can have any string as a value. Together they form a name/value pair that your coordination logic is responsible for processing.

For example, imagine an FTP endpoint defined by the `port` element in [Example 111](#).

Example 111: *FTP Endpoint with Custom Properties*

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

The endpoint is configured using two custom FTP properties:

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

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

Supporting the properties

The `initialize()` method of both the client-side filename factory and the server-side filename factory take a

`com.iona.webservices.wsdl.ext.ftp.FTPProperties` object. The `FTPProperties` object is populated by the contents of the endpoints `ftp:properties` element when the transport is initialized.

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

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

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

`com.iona.webservices.wsdl.ext.ftp.FTPProperty.NAME`. You can extract the value of the `value` attribute using the `FTPProperty` object's `getProperty()` with the constant

`com.iona.webservices.wsdl.ext.ftp.FTPProperty.VALUE`. Once you have the values of the properties, it is up to you to determine how they impact the coordination scheme.

[Example 112](#) shows code for supporting the properties shown in [Example 111 on page 239](#).

Example 112: Using Custom FTP Properties

```
import com.iona.webservices.wsdl.ext.FTPProperties;
import com.iona.webservices.wsdl.ext.FTPProperty;

String nameTypeProp = "UseHumanNames";
String lastNameProp = "LastName";
for (Iterator it = properties.getExtensors(); it.hasNext();)
{
    FTPProperty property = (FTPProperty)it.next();
    String n = property.getProperty(FTPProperty.NAME);
```

Example 112:*Using Custom FTP Properties*

```

if (nameTypeProp.equals(n))
{
    Boolean useHuman = new
    Boolean(property.getProperty(FTPProperty.VALUE));
}

if (lastNameProp.equals(n))
{
    String lastName = property.getProperty(FTPProperty.VALUE);
}
}

```

**Filling in the Filename Factory
Property Metadata**

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

`FilenameFactoryPropertyMetaData` objects have three fields:

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

`FilenameFactoryPropertyMetaData` objects do not have any methods for populating its fields once the object is instantiated. You must set all of the values using the constructor that is shown in [Example 113](#).

Example 113:*Constructor for FilenameFactoryPropertyMetaData*

```

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

```

[Example 114](#) shows code for creating an array to be returned from `getPropertiesMetaData()`.

Example 114:*Populating the Filename Properties Metadata*

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

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

Part IV

Other Artix Features

In this part

This part contains the following chapters:

Working with CORBA	page 245
Adding Routing Instructions	page 259
Using the Artix Transformer	page 297
Using Codeset Conversion	page 319

Working with CORBA

Artix provides extensions for describing CORBA applications as services

Overview

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

In the chapter

This chapter discusses the following topics:

Adding a CORBA Binding	page 246
Creating a CORBA Endpoint	page 252

Adding a CORBA Binding

Overview

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

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

Options

To add a CORBA binding to an Artix contract you can choose one of four methods. The first option is to use the Artix Designer. The Designer provides a wizard that automatically generates the binding and type map information for a specified port type.

The second option is to use the `wsdltocorba` command line tool. The command line tool automatically generates the binding and type map information for a specified port type. See [“Using wsdltocorba” on page 247](#).

The third option is to enter the binding and typemap information by hand using a text editor or XML editor. This option provides you the flexibility to customize the binding. However, hand editing Artix contracts can be a time consuming process and provides no error checking mechanisms. For information on the WSDL extensions used to specify a CORBA binding see [“Mapping to the binding” on page 248](#).

Using `wsdltocorba`

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

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

The command has the following options:

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

The generated WSDL file will also contain a CORBA port with no address specified. To complete the port specification you can do so manually or use Artix Designer.

WSDL Namespace

The WSDL extensions used to describe CORBA data mappings and CORBA transport details are defined in the WSDL namespace

`http://schemas.ionas.com/bindings/corba`. To use the CORBA extensions you will need to include the following in the `definitions` tag of your contract:

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

Mapping to the binding

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

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

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

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

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

would produce the following `corba:binding`:

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

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

corba:param is a child of `corba:operation`. Each `part` element of the input and output messages specified in the logical operation, except for the part representing the return value of the operation, must have a corresponding

`corba:param` element. The parameter order defined in the binding must match the order specified in the IDL definition of the operation. The `corba:param` element has the following required attributes:

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

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

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

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

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

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

Example

For example, a logical interface for a system to retrieve employee information might look similar to `personalInfoLookup`, shown in [Example 115](#).

Example 115: *personalInfo lookup port type*

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

The CORBA binding for `personalInfoLookup` is shown in [Example 116](#).

Example 116:*personalInfoLookup CORBA Binding*

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

Creating a CORBA Endpoint

Overview

Generally, when you are creating a CORBA endpoint with Artix, you need to do two things. First, you must specify the port information in the Artix contract so that Artix can instantiate the appropriate port. Second, you must generate the IDL describing your service so that a native CORBA application can understand the interfaces of the new service.

In this section

This section discusses the following topics:

Configuring an Artix CORBA Endpoint	page 253
Generating CORBA IDL	page 257

Configuring an Artix CORBA Endpoint

Overview

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

Namespace

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

CORBA address specification

The IOR of the CORBA object is specified using the `corba:address` element. You have four options for specifying IORs in Artix contracts:

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

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

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

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

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

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

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

For more information on using the name service with Artix see [Artix for CORBA](#).

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

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

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

Specifying POA policies

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

- [POA Name](#)
- [Persistence](#)
- [ID Assignment](#)

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

POA Name

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

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

Persistence

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

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

ID Assignment

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

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

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

Using the command line tool

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

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

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

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

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to [Example 117](#):

Example 117: *CORBA personalInfoLookup Port*

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

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

Generating CORBA IDL

Overview

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

From the command line

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

```
wsdltocorba -idl -b binding [-corba] [-i portType] [-d dir]
             [-o file] [-L file] [-q] [-h] [-V] wSDL_file
```

The command has the following options:

<code>-idl</code>	Instructs the tool to generate an IDL file from the specified binding.
<code>-b <i>binding</i></code>	Specifies the CORBA binding from which IDL is to be generated.
<code>-corba</code>	Instructs the tool to generate a CORBA binding for the specified port type.
<code>-i <i>portType</i></code>	Specifies the name of the port type being mapped to a CORBA binding.
<code>-d <i>dir</i></code>	Specifies the directory into which the new WSDL file is written.
<code>-o <i>file</i></code>	Specifies the name of the generated WSDL file. Defaults to <code>wSDL_file.idl</code> .
<code>-L <i>file</i></code>	Specifies the location of your Artix license file. The default behavior is to check <code>IT_PRODUCT_DIR\etc\license.txt</code> .
<code>-q</code>	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
<code>-h</code>	Specifies that the tool will display a usage message.
<code>-v</code>	Specifies that the tool runs in verbose mode.

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

Adding Routing Instructions

Artix provides message routing based on operations, ports, message attributes, or message content.

In this chapter

This chapter discusses the following topics:

Artix Routing	page 260
Compatibility of Ports and Operations	page 261
Defining Routes in Artix Contracts	page 264
Creating Routes from the Command Line	page 279
Advanced Routing Features	page 282
Error Handling	page 291
Service Lifecycles	page 292
Routing References to Transient Servants	page 294

Artix Routing

Overview

Artix routing is implemented by the Artix routing service and is controlled by rules specified in the routing service's contract. Artix routers can be deployed either in standalone mode or embedded into another Artix endpoint.

Artix supports the following types of routing:

- [Port-based](#)
- [Operation-based](#)
- [Attribute-based](#)
- [Content-based](#)

A router's contract must include definitions for the source services and destination services. The contract also defines the routes that connect the source endpoints to the destination endpoints. This routing information is all that is required to implement routing.

Port-based

Port-based routing acts on the port or transport-level identifier, specified by a `port` element in an Artix contract. Messages are routed based the endpoint that sends them. This is the most efficient form of routing.

Operation-based

Operation-based routing lets you route messages based on the logical operations described in an Artix contract. Messages are routed between logical operations whose arguments are equivalent. Operation-based routing can be specified on the logical interface, defined in a `portType` element, level or the finer grained logical operation level.

Attribute-based

Attribute-based routing lets you route messages based on the values set in the transport header of a message. Messages are routed to a particular endpoint based on attributes set in the message's transport header.

Content-based

Content-based routing lets you route messages based on the content of the request message. Messages are routed to a particular endpoint based on the message's content. The routes are defined using simple XPATH expressions that query the message content and select a destination based on the result.

Compatibility of Ports and Operations

Overview

Artix can route messages between endpoints that expect similar messages. The endpoints can use different message transports and different payload formats, but the messages must be logically identical. For example, if you have a baseball scoring service that is hosted on a mainframe, it might send data using fixed record length fields over a WebSphere MQ queue. Using Artix, you can route the score data to a reporting service that consumes SOAP data over HTTP. The only requirement for operation-based routing is that the two services exposed by the endpoints have a logical operation that uses messages with the same logical description. For port-based routing, the destination endpoint must have a matching logical operation defined for each of the logical operations defined for the source endpoint.

Port-based routing

Port-based routing is rough grained in that the routing rules are defined on the `port` elements of an Artix contract and do not look at the individual logical operations defined in the logical interface, defined by a `portType` element, for which the `port` element defines an endpoint. Therefore, port-based routing requires that the endpoints between which messages are routed must have compatible logical interface descriptions.

For two endpoints to have compatible logical interfaces the following conditions must be met:

- The `portType` element defining the destination's logical interface must contain a matching `operation` element for each `operation` element in the `portType` element defining the source's logical interface. Matching `operation` elements must have the same value in their `name` attribute.
- Each of the matching `operation` elements must have the same number of `input`, `output`, and `fault` elements.
- Each of the matching `operation` elements' `input` elements must be associated to a logical message, defined by a `message` element, whose sequence of `part` elements have matching types.
- Each of the matching `operation` elements' `output` elements must be associated to a logical message whose sequence of `part` elements have matching types.

- Each of the matching `operation` elements' `fault` elements must be associated to a logical message whose sequence of `part` elements have matching types.

For example, given the two logical interfaces defined in [Example 118](#) you could construct a route from an endpoint bound to `baseballScorePortType` to an endpoint bound to `baseballGamePortType`. However, you could not create a route from an endpoint bound to `finalScorePortType` to an endpoint bound to `baseballGamePortType` because the message types used for the `getScore` operation do not match.

Example 118: *Logical interface compatibility example*

```
<message name="scoreRequest">
  <part name="gameNumber" type="xsd:int"/>
</message>
<message name="baseballScore">
  <part name="homeTeam" type="xsd:int"/>
  <part name="awayTeam" type="xsd:int"/>
  <part name="final" type="xsd:boolean"/>
</message>
<message name="finalScore">
  <part name="home" type="xsd:int"/>
  <part name="away" type="xsd:int"/>
  <part name="winningTeam" type="xsd:string"/>
</message>
<message name="winner">
  <part name="winningTeam" type="xsd:string"/>
</message>
<portType name="baseballGamePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:baseballScore" name="baseballScore"/>
  </operation>
  <operation name="getWinner">
    <input message="tns:scoreRequest" name="winnerRequest"/>
    <output message="tns:winner" name="winner"/>
  </operation>
</portType>
<portType name="baseballScorePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:baseballScore" name="baseballScore"/>
  </operation>
</portType>
```

Example 118: *Logical interface compatibility example*

```
<portType name="finalScorePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:finalScore" name="finalScore"/>
  </operation>
</portType>
```

Operation-based routing

Operation-based routing rules check for compatibility based on the `operation` elements of the logical interface description. Therefore, messages can be routed between any two compatible logical operations.

The following conditions must be met for operations to be compatible:

- The operations must have the same number of `input`, `output`, and `fault` elements.
- The logical messages must have the same sequence of part types.

For example, if you added the logical interface in [Example 119](#) to the interfaces in [Example 118 on page 262](#), you could specify a route from `getFinalScore` defined in `fullScorePortType` to `getScore` defined in `finalScorePortType`. You could also define a route from `getScore` defined in `fullScorePortType` to `getScore` defined in `baseballScorePortType`.

Example 119: *Operation-based routing interface*

```
<portType name="fullScorePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:basballScore" name="baseballScore"/>
  </operation>
  <operation name="getFinalScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:finalScore" name="finalScore"/>
  </operation>
</portType>
```

Defining Routes in Artix Contracts

Overview

Artix routing is fully implemented by a contract that defines the integration of your systems. Routes are defined using WSDL extensions that are defined in the namespace `http://schemas.ionac.com/routing`. The most commonly used of these extensions are:

routing:route is the root element of any route defined in the contract.

routing:source specifies the port that acts as the source for messages that are to be routed.

routing:destination specifies the port to which messages will be routed. You do not need to do any programming and your applications need not be aware that any routing is taking place.

In this section

This section discusses the following topics:

Using Port-Based Routing	page 265
Using Operation-Based Routing	page 268
Using Attribute-Based Routing	page 271
Using Content-Based Routing	page 274

Using Port-Based Routing

Overview

Port-based routing is the easiest type of routing to implement. All of the rules are specified in an Artix contract describing how your endpoints are integrated. The routes specify the source endpoint for the messages and the destination endpoint to which messages are routed.

Describing routes in an Artix contract

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

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

To describe a port-based route you use three elements:

routing:route

The `routing:route` element is the root element of each route you describe in your contract. It takes on required attribute, `name`, that specifies a unique identifier for the route. The `routing:route` element also has an optional attribute, `multiRoute`, which is discussed in [“Advanced Routing Features” on page 282](#).

routing:source

The `routing:source` element specifies the endpoint from which the route redirects messages. A route can have several `routing:source` elements as long as they all meet the compatibility rules for port-based routing discussed in [“Port-based routing” on page 261](#).

The `routing:source` element requires two attributes: `service` and `port`. The `service` attribute specifies the `service` element in which the source endpoint is defined. The `port` attribute specifies the name of the `port` element defining the endpoint from which messages are being received. The router will create a proxy to listen for messages on this endpoint.

routing:destination

The `routing:destination` element specifies the endpoint to which the source messages are directed. The destination endpoint must be compatible with the source endpoint. For a discussion of the compatibility rules for port-based routing see [“Port-based routing” on page 261](#).

In standard routing only one destination is allowed per route. Multiple destinations are allowed in conjunction with the `routing:route` element’s `multiRoute` attribute that is discussed in [“Advanced Routing Features” on page 282](#).

The `routing:destination` requires two attributes: `service` and `port`. The `service` attribute specifies the `service` element in which the destination endpoint is defined. The `port` attribute specifies the name of the `port` element defining the endpoint to which messages are being sent.

Example

For example, to define a route from `baseballScorePortType` to `baseballGamePortType`, defined in [Example 118 on page 262](#), your Artix contract would contain the elements in [Example 120](#).

Example 120:Port-based routing example

```

1 <service name="baseballScoreService">
  <port binding="tns:baseballScoreBinding"
    name="baseballScorePort">
    <soap:address location="http://localhost:8991"/>
  </port>
</service>
<service name="baseballGameService">
  <port binding="tns:baseballGameBinding"
    name="baseballGamePort">
    <tibrv:port serverSubject="com.mycompany.baseball"/>
  </port>
</service>
2 <routing:route name="baseballRoute">
  <routing:source service="tns:baseballScoreService"
    port="tns:baseballScorePort"/>
  <routing:destination service="tns:baseballGameService"
    port="tns:baseballGamePort"/>
</routing:route>

```

There are two sections to the contract fragment shown in [Example 120](#):

1. The logical interfaces must be bound to physical ports in `service` elements of the Artix contract.
2. The route, `baseballRoute`, is defined with the appropriate `service` and `port` attributes.

Using Operation-Based Routing

Overview

Operation-based routing is a refinement of port-based routing. With operation-based routing you can specify specific logical operations within a logical interface as a source or a destination.

Describing routes in an Artix contract

The contract elements for defining operation-based routes are defined in the same namespace as the elements for port-based routing and you will need to include it in your contract's namespace declarations to use operation-based routing.

To specify an operation-based route you need to specify one additional element in your route description: `routing:operation`. The `routing:operation` element takes one required attribute, `name`, that specifies the value of the `name` attribute of an `operation` element in the source endpoint's logical interface. This `operation` element becomes the source of messages that are routed. Messages corresponding to other logical operations will not be routed.

The `routing:operation` element also has one optional attribute, `target`, that specifies the value of the `name` attribute of an `operation` element in the destination endpoint's logical interface. This `operation` element becomes the destination of messages redirected by the route. If a `target` is specified, messages are routed between the two operations. If no `target` is specified, the source operation's name is used as the name of the target operation. The source and target operations must meet the compatibility requirements discussed in [“Operation-based routing” on page 263](#).

You can specify any number of `routing:operation` elements in a route. They must be specified after all of the `routing:source` elements and before any `routing:destination` elements.

How operation-based rules are applied

Operation-based routing rules apply to all of the `routing:source` elements in the route. Therefore, if an operation-based routing rule is specified, a message will be routed if all of the following are true:

- The message is received from one of the endpoints specified in a `routing:source` element.

- The operation name associated with the received message is specified in one of the `routing:operation` elements.

If there are multiple operation-based rules in the route, the message will be routed to the destination specified in the matching operation's `target` attribute.

Example

For example to route messages from `getFinalScore` defined in `fullScorePortType`, shown in [Example 119 on page 263](#), to `getScore` defined in `finalScorePortType`, shown in [Example 118 on page 262](#), your Artix contract would contain the elements in [Example 121](#).

Example 121: Operation to Operation Routing

```

1 <service name="fullScoreService">
  <port binding="tns:fullScoreBinding"
    name="fullScorePort">
    <mq:server QueueManager="BBQM"
      QueueName="MLBQueue"
      ReplyQueueManager="BBRQM"
      ReplyQueueName="MLBScoreQueue"/>
  </port>
</service>
<service name="finalScoreService">
  <port binding="tns:finalScoreBinding"
    name="finalScorePort">
    <soap:address location="http://artie.com/finalScoreServer"/>
  </port>
</service>
2 <routing:route name="scoreRoute">
  <routing:source service="tns:fullScoreService"
    port="tns:fullScorePort"/>
  <routing:operation name="getFinalScore" target="getScore"/>
  <routing:destination service="tns:finalScoreService"
    port="tns:finalScorePort"/>
</routing:route>

```

There are two sections to the contract fragment shown in [Example 121](#):

1. The logical interfaces must be bound to physical endpoints in `service` elements of the Artix contract.
2. The route, `scoreRoute`, is defined using the `routing:operation` element.

You could also create a route between the operation `getScore`, defined in the logical interface `baseballGamePortType`, and an endpoint bound to `baseballScorePortType`. See [Example 118 on page 262](#). The resulting contract would include the fragment shown in [Example 122](#).

Example 122: *Operation to Port Routing Example*

```
<service name="baseballGameService">
  <port binding="tns:baseballGameBinding"
        name="baseballGamePort">
    <soap:address location="http://localhost:8991"/>
  </port>
</service>
<service name="baseballScoreService">
  <port binding="tns:baseballScoreBinding"
        name="baseballScorePort">
    <iiop:address location="file:\\score.ref"/>
  </port>
</service>
<routing:route name="scoreRoute">
  <routing:source service="tns:baseballGameService"
                port="tns:baseballGamePort"/>
  <routing:operation name="getScore"/>
  <routing:destination service="tns:baseballScoreService"
                    port="tns:baseballScorePort"/>
</routing:route>
```

Note that the `routing:operation` element only uses the `name` attribute. In this case the logical interface bound to `baseballScorePort`, `baseballScorePortType`, must contain an operation `getScore` that has matching messages as discussed in [“Port-based routing” on page 261](#).

Using Attribute-Based Routing

Overview

Artix allows you to route messages based on the transport attributes set in a message's header when using HTTP or WebSphere MQ. You can also route messages based on security settings and the CORBA principle.

Defining the rules

Rules based on message header transport attributes are defined in `routing:transportAttribute` elements in the route definition. Transport attribute rules are defined after all of the operation-based routing rules and before any destinations are listed.

The criteria for determining if a message meets the transport attribute rule are specified in sub-elements of the `routing:transportAttribute` element. A message passes the rule if it meets each criterion specified in the listed sub-element.

Each sub-element has a `contextName` attribute to specify the context in which the attribute is defined and a `contextAttributeName` attribute to specify the name of the attribute to be evaluated. The `contextName` attribute is specified using the QName of the context in which the attribute is defined. The contexts shipped with Artix are described in [Table 29](#). The `contextAttributeName` is also a QName and is relative to the context specified. For example, `UserName` is a valid attribute name for any of the HTTP contexts, but not for the MQ contexts.

Table 29: *Context QNames*

Context QName	Details
<code>http-conf:HTTPServerIncomingContexts</code>	Contains the attributes for HTTP messages being received by a service.
<code>corba:corba_input_attributes</code>	Contains the data stored in the CORBA principle.
<code>mq:IncomingMessageAttributes</code>	Contains the attributes for MQ messages being received by a service.

Table 29: *Context QName*

Context QName	Details
bus-security	Contains the attributes used by the IONA security service to secure your services.

Most sub-elements have a `value` attribute that can be tested. When dealing with string comparisons all elements have an optional `ignorecase` attribute that can have the values `yes` or `no` (`no` is the default). Each of the sub-elements can occur zero or more times, in any order:

routing:equals applies to string or numeric attributes. For strings, the `ignorecase` attribute may be used.

routing:greater applies only to numeric attributes and tests whether the attribute is greater than the value.

routing:less applies only to numeric attributes and tests whether the attribute is less than the value.

routing:startswith applies to string attributes and tests whether the attribute starts with the specified value.

routing:endswith applies to string attributes and tests whether the attribute ends with the specified value.

routing:contains applies to string or list attributes. For strings, it tests whether the attribute contains the value. For lists, it tests whether the value is a member of the list. The `contains` element accepts the optional `ignorecase` attribute for both strings and lists.

routing:empty applies to string or list attributes. For lists, it tests whether the list is empty. For strings, it tests for an empty string.

routing:nonempty applies to string or list attributes. For lists, it passes if the list is not empty. For strings, it passes if the string is not empty.

For information on the transport attributes for HTTP see [“Using HTTP” on page 161](#). For information on the transport attributes for WebSphere MQ see [“Using WebSphere MQ” on page 187](#).

Example

Example 123 shows a route using transport attribute rules based on HTTP header attributes. Only messages sent to the server whose `UserName` is equal to `JohnQ` will be passed through to the destination port.

Example 123: Transport Attribute Rules

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

Using Content-Based Routing

Overview

Content-based routing allows you to redirect messages based on their content. For example, you could route purchase orders based on their total amount or on their department identification. These routes are specified using XPath expressions that select a piece of a message to inspect.

Procedure

To create a content-based route in your contract you need to do the following things:

1. Add an XPath expression to select message content using a `routing:expression` element.
 2. Add a new route to your contract using a `routing:route` element.
 3. Add a source endpoint to your route using a `routing:source` element.
 4. Specify the expression to use as a routing criteria using a `routing:query` element.
 5. Add one or more `routing:destination` elements as children to the `routing:query` element.
 6. If you want to add a default destination endpoint, add a `routing:destination` element as a child of the `routing:route` element.
-

Router message format

Before you can write an XPath expression to select content from a message passing through the router, you need to understand how the router sees the message. The router sees the message as a virtual XML document that is reconstructed from the WSDL definitions in the contract. The mapping is done as follows:

1. The name of the message's root element is the QName of the `message` element referred to by the operation's `input` element.
2. Each `part` element of the message referenced by the `input` element is mapped to an element derived from the `name` attribute of the `part` element.
3. If the `part` element is of a complex type, or an element of a complex type, the type's elements appear inside of the element corresponding to the `part` element.

For example, if you had a service defined by the WSDL fragment in [Example 124](#) and were going to route requests to the register operation, the router would scan an XML document constructed using the message `ticketRequest`, which is the input message.

Example 124: *WSDL Fragment*

```
<definitions targetNamespace="vehicle.demo.example"
             xmlns:tns="vehicle.demo.example"
             ...>
  <types ...>
    ...
    <complexType name="vehicleType">
      <element name="vin" type="xsd:string" />
      <element name="model" type="xsd:string" />
    </complexType>
    ...
  </types>
  ...
  <message name="ticketRequest">
    <part name="vehicle" type="xsd:vehicleType"/>
    <part name="name" type="xsd:string"/>
    <part name="parkTime" type="xsd:string" />
  </message>
  ...
  <portType name="parkingLotMeter">
    <operation name="register">
      <input name="parkedCar" message="tns:ticketRequest"/>
      ...
    </operation>
    ...
  </portType>
  ...
</definitions>
```

When the router reconstructs the message, it the input message's name, given in the `input` element, as the name of the XML document's root element. It uses the message parts and the schema types to recreate the remaining elements in the XML document. The resulting XML document would look like [Example 125](#).

Example 125:*Router Message*

```
<ns1:parkedCar xmlns:ns1="vehicle.demo.example">
  <vehicle>
    <VIN>0123456789</VIN>
    <model>Prius</model>
  </vehicle>
  <name>Old MacDonald</name>
  <time>19:00</time>
</ns1:parkedCar>
```

Writing XPath expressions

XPath is a standard grammar for addressing the parts of an XML document. The Artix router uses XPath expressions to extract the content of a message for use in content-based routing. For example, if you wanted to write an XPath expression to extract the data stored in the `model` element of the XML document in [Example 125](#) you could use the XPath expression `parkedCar\vehicle\model` which translates into select the `model` element whose parent is a `vehicle` element and has a `parkedCar` element as a parent.

You could also use the XPath expression `\\model` which translates into select all of the `model` elements that are a descendent of the root element. If there were multiple `model` elements, the expression would select them all and return a string representing the nodeset of `model` elements.

Adding expressions to a contract

You add an XPath expression to your contract using a `routing:expression` element. The `routing:expression` element has two required attributes: `name` and `evaluator`. The `name` attribute specifies a unique identifier by which the expression is referred to when used in a route definition. The `evaluator` attribute specifies the type of expression being used to select the content.

Note: In Artix 4.0, XPath is the only supported grammar and is specified using the string `xpath`.

[Example 126](#) shows an example of adding an XPath expression to an Artix contract.

Example 126: *Expression in an Artix Contract*

```
<routing:expression name="widgetSize" evaluator="xpath">
  /*/widgetOrderform/type
</routing:expression>
```

The XPath expression selects the `type` child element of the `widgetOrderForm` element in the message. The `widgetOrderForm` element is not the root element of the message. It is generated from one of the `part` elements defined in the contract.

For more information on XPath see the specification at <http://www.w3.org/TR/xpath> or see the tutorial at <http://www.w3schools.com/xpath>.

Using expressions in a route

To use the expression to route messages, you need to add it to the route. This is done using the `routing:query` element. The `routing:query` element is a child of the `routing:route` element and must follow a single `routing:source` element. It has one attribute, `expression`, that specifies the name of the expression used to select a destination endpoint.

Specifying destinations for a content based route

The destinations that can be selected by the expression are specified using `routing:destination` elements that are children of the `routing:query` element. When used in content-based routes, the `routing:destination` elements use the `id` attribute. The `id` attribute specifies the value of the expression that will select the destination endpoint.

For example, the route shown in [Example 127](#) specifies a content-based route that uses the expressing defined in [Example 126](#) and has three possible destination endpoints.

Example 127:*Content-Based Route*

```
<routing:route name="sizeRoute">
  <routing:source service="tns:orderService" />
  <routing:query expression="tns:widgetSize">
    <routing:destination id="small" service="tns:smallService" />
    <routing:destination id="med" service="tns:medService" />
    <routing:destination id="big" service="tns:bigService" />
  </routing:query>
</routing:route>
```

If the value of the message's `type` element is `med`, the message will be routed to the endpoint defined by the contract's `service` element whose name attribute equals `medService`.

Adding a default destination

To add a default destination for a content based route, you simply add a `routing:destination` element after the `routing:query` element. If none of the destination endpoints specified by the content-based route are selected, the first destination after the `routing:query` element is selected.

[Example 128](#) shows a content-based route with a default destination endpoint.

Example 128:*Content-Based Route with a Default Destination*

```
<routing:route name="sizeRoute">
  <routing:source service="tns:orderService" />
  <routing:query expression="tns:widgetSize">
    <routing:destination id="small" service="tns:smallService" />
    <routing:destination id="med" service="tns:medService" />
    <routing:destination id="big" service="tns:bigService" />
  </routing:query>
  <routing:destination service="tns:miscService" />
</routing:route>
```

Creating Routes from the Command Line

Overview

If you do not wish to use the Artix Designer or want to add routes to contracts as part of a makefile, you can use the `wsdltorouting` command line tool. `wsdltorouting` will import an existing contract and generate a new contract containing the specified routing instructions. The imported contract must contain the specified source endpoint and destination endpoint, otherwise the tool will generate an error.

Usage

To generate a route using the command line tool, use the following command.

```
wsdltorouting [-rn name] [-ssn service] [-spn port]
              [-dsn service] [-dpn port] [-on operation]
              [-ta attribute] [-d dir] [-o file]
              [-L file] [-quiet] [verbose] [-h] [-v] wsdurl
```

`wsdltorouting` has the following options.

<code>-rn name</code>	Specifies the name of the generated route. If no name is given a unique name will be generated for the route.
<code>-ssn service</code>	Specifies the name of the <code>service</code> element to use as the source of the route.
<code>-spn port</code>	Specifies the name of the <code>port</code> element to use as the source of the route.
<code>-dsn service</code>	Specifies the name of the <code>service</code> element to use as the destination of the route.
<code>-dpn port</code>	Specifies the name of the <code>port</code> element to use as the destination of the route.
<code>-on operation</code>	Specifies the name of the operation to use for the route. If the route is port-based, you do not need to use this flag.
<code>-ta attribute</code>	Specifies a transport attribute to use in defining the route. For details on how to specify the transport attributes, see “Specifying transport attributes” on page 280 .
<code>-d dir</code>	Specifies the output directory for the generated contract.
<code>-o file</code>	Specifies the filename of the generated contract.

<code>-L file</code>	Specifies the location of your Artix license file. The default behavior is to check <code>IT_PRODUCT_DIR\etc\license.txt</code> .
<code>-quiet</code>	Specifies that the tool runs in quiet mode.
<code>-verbose</code>	Specifies that the tool runs in verbose mode.
<code>-h</code>	Displays the tool's usage statement.
<code>-v</code>	Displays the tool's version.

Specifying transport attributes

When using `wsdlto routing`, transport attributes are specified using four comma-separated values. The first value specifies the name of the attribute's context. The second value specifies the name of the attribute. The third value is the condition used to evaluate the attribute. The fourth value is the values against which the attribute is evaluated.

[Table 30](#) shows the valid context names to use in specifying a transport attribute.

Table 30: *Context Names Used with `wsdlto routing`*

Context Name	Artix Context
<code>HTTP_SERVER_INCOMING_CONTEXTS</code>	HTTP properties received as part of a client request
<code>CORBA_CONTEXT_ATTRIBUTES</code>	CORBA transport properties
<code>SECURITY_SERVER_CONTEXT</code>	Properties used to configure security settings

For more information on the properties available in the contexts see either [Developing Artix Applications in C++](#) or [Developing Artix Applications in Java](#).

[Table 31](#) shows the valid condition entries used in specifying transport attributes when using `wsdlto routing`.

Table 31: *Conditions Used with `wsdlto routing`*

Condition	WSDL Equivalent
<code>equals</code>	<code>routing>equals</code>
<code>startswith</code>	<code>routing:startswith</code>

Table 31: *Conditions Used with wsdltorouting*

Condition	WSDL Equivalent
endswith	routing:endswith
contains	routing:contains
empty	routing:empty
nonempty	routing:nonempty
greater	routing:greater
less	routing:less

Example

If you had a contract that contained the services `itchy` and `scratchy`, both with an equivalent operation `gouge`, you could use the command shown in [Example 129](#) to add a route to your contract.

Example 129: *Adding a Route with wsdltorouting*

```
wsdltorouting -rn itchyGougeScratchy -ssn itchy -spn gougerPort
-dsn scratchy -dpm gougedPort -on gouge
-ta HTTP_SERVER_INCOMING_CONTEXTS,UserName,equal,Goering
itchyscratchy.wsdl
```

The resulting route is shown in [Example 130](#).

Example 130: *Route from wsdltorouting*

```
<routing:route name="itchyGougeScratchy">
  <routing:source service="tns:itchy"
    port="tns:gougerPort"/>
  <routing:operation name="gouge"/>
  <routing:transportAttributes>
    <routing:equals
      contextName="http-conf:HTTPServerIncomingContexts"
      contextAttributeName="UserName"
      value="Goering"/>
    </routing:transportAttributes>
  <routing:destination service="tns:scratchy"
    port="gougedPort"/>
</routing:route>
```

Advanced Routing Features

Overview

Artix routing also supports the following advanced routing capabilities:

- Linking routes to build complex routing scenarios.
- Load balancing between a number of endpoints offering the same service.
- Broadcasting a message to a number of destinations.
- Specifying a failover service to route messages to provide a level of high availability.

All of these features use the optional `multiRoute` attribute on the `routing:route` element.

In this section

This section discusses the following topics:

Load Balancing	page 283
Message Broadcasting	page 284
Failover Routing	page 286
Linking Routes	page 288

Load Balancing

Overview

The router can load balance requests across a number of endpoints without requiring any special configuration or programming. It uses a round-robin algorithm to route requests, that match a routing rule, to one of the specified destination endpoints.

Specifying router based load balancing

Router-based load balancing rules are defined using the `routing:route` element's `multiRoute` attribute. To define a failover route you set the `multiRoute` attribute to `loadBalance`. Within the route definition you define a message source as you would for any other route. You also specify a number of destination endpoints to which messages will be routed. Using a round-robin algorithm the router will direct each request from the source endpoint to one of the specified destination endpoints.

Example

For example, if you had three endpoints that could process requests for baseball scores and wanted to balance the request load among them, you could create a route similar to the one shown in [Example 131](#).

Example 131: Router Based Load Balancing

```
<routing:route name="scoreRoute" multiRoute="loadBalance">
  <routing:source service="tns:baseballGameService"
    port="tns:baseballGamePort"/>
  <routing:operation name="getScore"/>
  <routing:destination service="tns:baseballScoreService1"
    port="tns:baseballScorePort"/>
  <routing:destination service="tns:baseballScoreService2"
    port="tns:baseballScorePort"/>
  <routing:destination service="tns:baseballScoreService3"
    port="tns:baseballScorePort"/>
</routing:route>
```

Using this route, each time a new request was received for the `getScore` operation, the router would direct it to whichever endpoint was next in the rotation. So, the first request would be routed to `baseballScoreService1`, the second request would be routed to `baseballScoreService2`, the third request would be routed `baseballScoreService3`, and so forth.

Message Broadcasting

Overview

Using the router, you can broadcast a message to multiple endpoints. For example, you could deploy an endpoint whose function is to generate shutdown warnings to all services deployed in a network. You could simplify the development of this service by using an Artix router to intercept a single warning message and broadcast it to the other services. In this way, you would only need to change the router's contract when you add or remove services.

Defining broadcasting rules

You define rules by setting the `multiRoute` attribute in the `routing:route` element to `fanout` in your route definition. This causes routed messages to be transmitted to all of the endpoints specified by the route's `routing:destination` elements.

There are three restrictions to using the fanout method of message broadcasting:

- All of the source endpoints and destination endpoints must be oneways. In other words, they cannot have any output messages.
- The source endpoints and destination endpoints cannot have any fault messages.
- The input messages of the source endpoints and destination endpoints must meet the compatibility requirements as described in [“Compatibility of Ports and Operations” on page 261](#).

Example

[Example 132](#) shows an Artix contract fragment describing a route for broadcasting a message to a number of endpoints.

Example 132:*Fanout Broadcasting*

```
<message name="statusAlert">
  <part name="alertType" type="xsd:int"/>
  <part name="alertText" type="xsd:string"/>
</message>
```

Example 132:*Fanout Broadcasting*

```

<portType name="statusGenerator">
  <operation name="eventHappens">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
<portType name="statusChecker">
  <operation name="eventChecker">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
<service name="statusGeneratorService">
  <port binding="tns:statusGeneratorBinding"
        name="statusGeneratorPort">
    <soap:address location="http://localhost:8081"/>
  </port>
</service>
<service name="statusCheckerService">
  <port binding="tns:statusCheckerBinding"
        name="statusCheckerPort1">
    <corba:address location="file://status1.ref"/>
  </port>
  <port binding="tns:statusCheckerBinding"
        name="statusCheckerPort2">
    <tuxedo:server>
      <tuxedo:service name="personalInfoService">
        <tuxedo:input operation="infoRequest"/>
      </tuxedo:service>
    </tuxedo:server>
  </port>
</service>
<routing:route name="statusBroadcast" multiRoute="fanout">
  <routing:source service="tns:statusGeneratorService"
                 port="tns:statusGeneratorPort"/>
  <routing:operation name="eventHappens" target="eventChecker"/>
  <routing:destination service="tns:statusCheckerService"
                      port="tns:statusCheckerPort1"/>
  <routing:destination service="tns:statusCheckerService"
                      port="tns:statusCheckerPort2"/>
</routing:route>

```

Failover Routing

Overview

The Artix router can provide a basic level of high-availability by allowing you to create routes that define failover scenarios. The router will automatically redirect messages to a new endpoint if the current destination fails. The router will attempt to send a request to all the destinations in a route before throwing an exception back to the client.

Defining the failover rules

To define a failover route you set the `routing:route` element's `multiRoute` attribute to `failover`. When you designate a route as failover, the routed message's target is selected using a round-robin algorithm. If the first target in the list is unable to receive the message, it is routed to the second target. The route will traverse the destination list until either one of the target services can receive the message or the end of the list is reached. On the next failure, the router will start searching from the last position on the list. So if the message was routed to the second entry on the list to deal with an initial failure, the router will start directing requests to the third entry on the list to handle the second failure. When the end of the list is reached, the router will start at the beginning again. If the router is unsuccessful in delivering a message after trying each service in the failover route once, the router will report that the message is undeliverable.

Example

Given the route shown in [Example 133](#), the message will first be routed to `destinationPortA`. If service on `destinationPortA` cannot receive the message, it is routed to `destinationPortB`.

Example 133:Failover Route

```
<routing:route name="failoverRoute" multiRoute="failover">
  <routing:source service="tns:sourceService"
    port="tns:sourcePort"/>
  <routing:destination service="tns:destinationServiceA"
    port="tns:destinationPortA"/>
  <routing:destination service="tns:destinationServiceB"
    port="tns:destinationPortB"/>
  <routing:destination service="tns:destinationServiceC"
    port="tns:destinationPortC"/>
</routing:route>
```

If `destinationPortB` fails at some future point, the messages are then routed to `destinationPortC`. If `destinationPortC` cannot receive messages, the router will then try `destinationPortA`. If `destinationPortA` is not available, the router will try `destinationPortB`. If `destinationPortB` is unavailable, the router will report that the message cannot be delivered.

Linking Routes

Overview

There are occasions, particularly when using content-based routing or using one of the multi-endpoint routing features, when you need to link together a number of routing criteria. Using the Artix router you can do this by linking together a number of routes. For example, you may want to route orders for customers in Brazil to a local endpoint, but you also want the orders to automatically fail-over to a alternative endpoint. You can do this by creating a content-based route that specifies a fail-over route as a destination.

Specifying a route as a destination

In Artix you link routes together by specifying one route as the destination of another route. When the destination specifying the linked route is selected, the message is passed through the second route to determine its destination. The second route may also contain destinations that contain linked routes. The message will pass through each linked route in order until a destination containing an endpoint is selected.

To specify a linked route as a destination you replace the `service` attribute and the `port` attribute in a `routing:destination` element with the `route` attribute. The value of the `route` attribute must correspond to the name of another route in the contract. The specified route becomes linked with the destination and any message that selects this destination will be processed through it.

Example

Imagine that your company had order processing centers in several cities and you needed to route orders to the processing center closest to the delivery address. You could implement this using a content-based route as shown in [Example 134](#).

Example 134:*Content-Based Route*

```
<routing:expression name="zipCode" evaluator="xpath">
  tns:placeWidgetOrder/widgetOrderForm/shippingAddress/zipCode
</routing:expression>
```

Example 134:*Content-Based Route*

```

<routing:route name="zipCodeRoute">
  <routing:source service="tns:widgetOrderService"
    port="tns:SOAPPort" />
  <routing:query expression="tns:zipCode">
    <routing:destination id="02452"
      service="tns:widgetOrderServiceEast"
      port="walthamPort" />
    <routing:destination id="91105"
      service="tns:widgetOrderServiceWest"
      port="passadenaPort" />
  </routing:query>
</routing:route>

```

If you needed to add a fail-over mechanism to ensure that the orders were processed by a different processing center in the event of a failure, you could simply add two linked routes for the destination of the content-based route as shown in [Example 135](#).

Example 135:*Linked Routes*

```

<routing:expression name="zipCode" evaluator="xpath">
  tns:placeWidgetOrder/widgetOrderForm/shippingAddress/zipCode
</routing:expression>
<routing:route name="walthamRoute" multiRoute="failover">
  <routing:destination service="tns:widgetOrderServiceEast"
    port="walthamPort" />
  <routing:destination service="tns:widgetOrderServiceWest"
    port="passadenaPort" />
</routing:route>
<routing:route name="passadenaRoute" multiRoute="failover">
  <routing:destination service="tns:widgetOrderServiceWest"
    port="passadenaPort" />
  <routing:destination service="tns:widgetOrderServiceEast"
    port="walthamPort" />
</routing:route>
<routing:route name="zipCodeRoute">
  <routing:source service="tns:widgetOrderService"
    port="tns:SOAPPort" />
  <routing:query expression="tns:zipCode">
    <routing:destination id="02452" route="tns:walthamRoute" />
    <routing:destination id="91105" route="tns:passadenaRoute" />
  </routing:query>
</routing:route>

```

[Example 135](#) expands on [Example 134](#) by adding two routes: `walthamRoute` and `passadenaRoute`. Both of these routes will not perform any routing on their own because they lack `routing:source` elements. They are instead used as destinations for the content-based route called `zipCodeRoute`. In [Example 134](#), the content-based route simply routed to one endpoint for each destination. In [Example 135](#), the route's destinations are linked routes. If the first destination is selected, the message is routed through the fail-over route `walthamRoute`. If the second destination is selected, the message is routed through the fail-over route `passadenaRoute`.

Error Handling

Initialization errors

Errors that can be detected during initialization while parsing the WSDL, such as routing between incompatible logical interfaces and some kinds of route ambiguity, are logged and an exception is raised. This exception aborts the initialization and shuts down the server.

Runtime errors

Errors that are detected at runtime are reported as exceptions and returned to the client; for example “no route” or “ambiguous routes”.

Service Lifecycles

Overview

When the Artix router uses dynamic proxy services, you can configure garbage collection of old proxies. Dynamic proxies are used when the router bridges endpoints that have patterns such as callback, factory, or any interaction that passes references to other endpoints. When the router encounters a reference in a message, it proxifies the reference into one that a receiving application can use. For example, an IOR from a CORBA server cannot be used by a SOAP client, so the router dynamically creates a new route for the SOAP client.

However, dynamic proxies persist in the router memory and can have a negative effect on performance. To overcome this, Artix provides service life cycle garbage collection, which cleans up old proxy services that are no longer used. This garbage collection service cleans up unused proxies when a threshold has been reached on a least recently used basis.

Configuring service lifecycle

To configure service garbage collection for the Artix router, perform the following steps:

1. Add the `service_lifecycle` plug-in to the `orb_plugins` list:

```
orb_plugins = ["xmlfile_log_stream", "service_lifecycle",  
              "routing"];
```

2. Configure the service lifecycle cache size:

```
plugins:service_lifecycle:max_cache_size = "30";
```

Writing client applications

When writing client applications, you must also make allowances for the garbage collection service; in particular, ensure that exceptions are handled appropriately.

For example, a client might attempt to proxy to an endpoint that has already been garbage collected. To prevent this, do either of the following:

- Handle the exception, get a new reference, and continue. However, in some cases, this might not be possible if the endpoint has state.
- Set `max_cache_size` to a reasonable limit to ensure that all your clients can be accommodated. For example, if you always expect to support 20 concurrent clients, each with a transient service session, you might wish to configure the `max_cache_size` to 30.

You do not want to impact any clients, and must ensure that an endpoint is no longer needed when it is garbage collected. However, if you set `max_cache_size` too high, this might use up too much router memory and have a negative impact on performance. For example, a suggested range for this setting is 30-100.

Routing References to Transient Servants

Overview

Applications create transient servants by cloning a `service` element defined in your contract. The cloned `service` element uses the same interface, binding, and transport as the `service` element in the contract. However, it has a unique QName and a unique address. So, a transient servant's `service` element only exists in the memory of the application that created it and possesses no link back to the `service` element from which it was cloned.

Because a transient servant does not have a `service` element in the physical contract and no link to one, the router, when it receives a reference to a transient servant, has no concrete information about how to create a proxy for the referenced servant. The router must make a best guess about which `service` element in its contract to use as the template for the proxy to the transient servant. To do this, the router chooses the first compatible `service` element in its contract.

Compatibility of services

A `service` element is considered compatible with a transient servant if it uses the same interface, binding, and transport as the transient servant. For example, if a transient servant was created using the `service` element, whose `name` attribute is set to `templateVendor`, in [Example 136](#) it would be compatible with `IIOPVendor`. However, it would not be compatible with `SOAPVendor` because `SOAPVendor` uses a different transport than `templateVendor`. Also, if `IIOPVendor` was defined using different transport properties, such as having a defined POA name, transient servants created from `templateVendor` would not be compatible.

Example 136: Contract with a Service Template

```
<definitions ...>
  ...
  <message name="mangoRequest">
    <part name="num" type="xsd:int"/>
  </message>
  <message name="mangoPrice">
    <part name="cost" type="xsd:float"/>
  </message>
```

Example 136: *Contract with a Service Template*

```

<portType name="fruitVendor">
  <operation name="sellMangos">
    <input name="num" message="tns:mangoRequest"/>
    <output name="price" message="tns:mangoPrice"/>
  </operation>
</portType>
<binding name="fruitVendorBinding" type="tns:fruitVendor">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sellMangos">
    <soap:operation soapAction="" style="rpc"/>
    <input name="num">
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://fruitVendor.com" use="encoded"/>
    </input>
    <output name="cost">
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://fruitVendor.com" use="encoded"/>
    </output>
  </operation>
</binding>
<service name="templateVendor">
  <port binding="tns:fruitVendorBinding"
    name="transientVendor">
    <iiop:address location="ior:"/>
  </port>
</service>
<service name="SOAPVendor">
  <port binding="tns:fruitVendorBinding"
    name="SOAPVendorPort">
    <soap:address location="localhost:5150"/>
  </port>
</service>
<service name="IIOPVendor">
  <port binding="tns:fruitVendorBinding"
    name="IIOPVendorPort">
    <iiop:address location="file:///objref.ior"/>
  </port>
</service>
</definitions>

```

Contract design issues

The router's means of selecting a compatible `service` element to create proxies for transient servants can result in odd behavior if you use the same interface to create both static servants and transient servants. When passing references to these servants through the router, the potential exists for the router to select the static servant to create proxies for the transient servants. When this happens, the router will silently redirect all of the messages to the servant defined by the static `service` element.

To avoid this situation be sure to place the `service` elements used to create transient servants before the `service` elements that will be used to create static servants. This will ensure that the router will find the transient servant's `service` elements.

Preventing bloat

Because the router creates a new proxy for each transient reference that passes through it, the router can suffer from memory bloating. To prevent bloating, you can specify two properties in the router's runtime configuration:

- the maximum number of proxified references the router maintains
- the maximum number of unproxified references the router maintains

The `plugins:routing:proxy_cache_size` variable specifies the number of proxified references the router maintains. The default is 50.

The `plugins:routing:reference_cache_size` variable specifies number of unproxified server references maintained by the router. The default is unbounded.

For example, take a SOAP-HTTP client and CORBA server banking system with 1,500 accounts. By default, the 50 most recently used accounts are present in the router as proxified references. The next 1450 most recently used are unproxified references.

For more information about configuring a router see [Configuring and Deploying Artix Solutions](#).

Using the Artix Transformer

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

In this chapter

This chapter discusses the following topics:

Using the Artix Transformer as a Service	page 298
Using Artix to Facilitate Interface Versioning	page 300
WSDL Messages and the Transformer	page 305
Writing XSLT Scripts	page 308

Using the Artix Transformer as a Service

Overview

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

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

Procedure

To use the Artix transformer as a service you:

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

Defining the server

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

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

After defining the logical details of the service, you need to define the binding and network details for the service. The transformer can use any of the bindings and transports supported by Artix. For information on adding a binding for the transformer read [“Understanding Bindings in WSDL” on page 43](#). For information on adding network details for the transformer read [“Understanding How Endpoints are Defined WSDL” on page 157](#).

Writing the scripts

The XSLT scripts tell the transformer what it needs to do to process the data it receives. The scripts can be as simple or complex as they need to be to perform the task. The only requirement is that they are valid XSLT documents. For more information about writing XSLT scripts read [“Writing XSLT Scripts” on page 308](#).

Configure the transformer

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

For a detailed discussion of how to configure and deploy the Artix Transformer see [Configuring and Deploying Artix Solutions](#).

Using Artix to Facilitate Interface Versioning

Overview

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

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

Procedure

To use the Artix transformer for interface versioning:

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

Creating a composite contract

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

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

For example, if you were creating a composite contract for interface versioning you would have two contracts; one for the server with the updated interface and one for the client using the legacy interface. The file name for the server's contract is `r2e2.wsdl` and the contract for the client is `r2e1.wsdl`. For simplicity, they are located in the same directory as the composite contract. The composite contract importing both versions of the interface is shown in [Example 137](#).

Example 137: Composite WSDL

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

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

Define the transformer's interface

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

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

Note: Fault messages are not supported.

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

For example, if the interface in [Example 137 on page 301](#) was updated so that both the client's request and the server's response need to be transformed the transformer's interface would need two operations. In this

example the name of the request message is `widgetRequest` and the name of the response message is `widgetResponse`. The interface for the transformer, `versionTransform`, is shown in [Example 138](#).

Example 138: *Versioning Interface*

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

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

Defining the physical details for the transformer

After defining the operations used in transforming between the different version of the interface, you need to define the binding and network details for the transformer. The transformer can use any of the bindings and transports supported by Artix. For information on adding a binding for the transformer read [“Understanding Bindings in WSDL” on page 43](#). For information on adding network details for the transformer read [“Understanding How Endpoints are Defined WSDL” on page 157](#).

Writing the XSLT scripts

The XSLT scripts tell the transformer what it needs to do to process the data it receives. The scripts can be as simple or complex as they need to be to perform the task. The only requirement is that they are valid XSLT documents. For more information about writing XSLT scripts read [“Writing XSLT Scripts” on page 308](#).

Configuring the transformer

The Artix transformer is an Artix plugin and can be loaded by an Artix process. This provides a great deal of flexibility in how you configure and deploy the process. For a detailed discussion of how to configure and deploy the Artix transformer see [Configuring and Deploying Artix Solutions](#).

Configuring a chain

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

For a detailed discussion of how to configure and deploy the Artix chain builder see [Configuring and Deploying Artix Solutions](#).

WSDL Messages and the Transformer

Overview

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

The incoming message

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

1. The name of the message's root element is the QName of the `message` element referred to by the operation's `input` element.
2. Each `part` of the `input` message is placed in an element derived from the `name` attribute of the `part`.
3. If the `part` is of a complex type, or an `element` of a complex type, the type's elements appear inside of the element containing the `part`.

For example, if you had a service defined by the WSDL fragment in [Example 139](#) and the transformer implemented the operation `configure` the XML document would be constructed using the message `oldClientInput`, which is the `input` message.

Example 139: WSDL Fragment for Transformer

```
<definitions targetNamespace="vehicle.demo.example"
  xmlns:tns="vehicle.demo.example"
  ...>
<types ...>
...
  <complexType name="vehicleType">
    <element name="vin" type="xsd:string" />
    <element name="model" type="xsd:string" />
  </complexType>
</types>
...
<message name="original">
  <part name="vehicle" type="xsd1:vehicleType"/>
  <part name="name" type="xsd:string"/>
</message>
```

Example 139: *WSDL Fragment for Transformer*

```

<message name="transformed">
  <part name="vehicle" type="xsd:string"/>
  <part name="firstName" type="xsd:string"/>
  <part name="lastName" type="xsd:string"/>
</message>
...
<portType name="parkingLotMeter">
  <operation name="configure">
    <input name="oldClientInput" message="tns:original"/>
    <output name="updatedInput" message="tns:transformed"/>
  </operation>
  ...
</portType>
...

```

When the message is reconstructed, the transformer uses the input message's name, given in the `input` element, as the name of the root element of the XML document. It then uses the message parts and the schema types to recreate the data as an XML message. So if the transformer was using the contract defined in [Example 139 on page 305](#) an input message processed by the transformer could look like [Example 140](#).

Example 140: *Transformer Input Message*

```

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

```

Output message

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

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

- The name of the message's root element is the QName of the message element referred to by the `output` element in the Artix contract.
- There are the same number of elements in the result as there are `part` elements in the output message definition.
- The elements in the result are based on the `name` attributes of the `part` elements in the output message definition.
- The data contained in the element representing the output message's `part` elements matched the XMLSchema definitions in the contract.

For example, a result message for the `configure` operation defined in [Example 139 on page 305](#) would look like [Example 141](#).

Example 141:*Transformer Output Message*

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

Writing XSLT Scripts

Overview

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

Procedure

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

To write a XSLT script you:

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

In this section

This section discusses the following topics:

Elements of an XSLT Script	page 309
XSLT Templates	page 311
Common XSLT Functions	page 317

Elements of an XSLT Script

Overview

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

The transform element

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

When creating an XSLT script you must set the version attribute to 1.0 to inform the transformer what version of XSLT you are using. In addition, you must provide an XML namespace shortcut for the XSLT namespace in the `<xsl:transform>` element. [Example 142](#) shows a valid `<xsl:transform>` element for an XSLT script.

Example 142:*XSLT Script Stylesheet Element*

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

Top level elements

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

- `xsl:import`
- `xsl:include`
- `xsl:strip-space`
- `xsl:preserve-space`
- `xsl:output`
- `xsl:key`
- `xsl:decimal-format`

- `xsl:namespace-alias`
- `xsl:attribute-set`
- `xsl:variable`
- `xsl:param`
- `xsl:template`

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

Example

[Example 143](#) shows a simple XSLT script that transforms `SSN` elements into `acctNum` elements.

Example 143: Simple XSLT Script

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

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

XSLT Templates

Overview

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

To write a template in XSLT:

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

`<xsl:template>` elements

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

Specifying source elements

You specify the elements of the source document to which template rules are matched using the `match` attribute of the `xsl:template` element. The source elements are specified using the syntax specified by the XPath specification (<http://www.w3.org/TR/xpath>). The source element address looks very similar to a file path where slash(/) specifies the root element and child elements are listed in top down order separated by a slash(/). For example to specify the `surname` element of the XML document shown in [Example 144](#), you would specify it as `/name/surname`.

Example 144: Sample XML Document

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

Template matching order

XSLT processors start processing with the `<xsl:template match="/">` element if it is present. All of the processing directives for this template act on the top-level elements of the source document. For example, given the XML document shown in [Example 144 on page 311](#) any processing rules specified in `<xsl:template match="/">` would apply to the `name` element. In addition, specifying a template for the root element(`/`) forces you to make all your source element paths explicit from the root element. The XSLT script shown in [Example 145](#) generates the string `Friday` when run on [Example 144 on page 311](#).

Example 145: XSLT Script with Root Element Template

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

You do not need to specify a template for the root element of the source document in an XSLT script. When you omit the root element's template the processor treats all template paths as though they originated from the source documents top level element. The XSLT script in [Example 146](#) generates the same output as the script in [Example 145](#).

Example 146: XSLT Script without Root Element Template

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

Template rules

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

script in [Example 147](#) takes an XML document with a `title` element and a `subTitle` element and produces an HTML document where the contents of `title` are displayed using the `<h1>` style and the contents of `subTitle` are displayed using the `<h2>` style.

Example 147: *XSLT Template with HTML*

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

Applying templates to child elements

You can instruct the XSLT processor to apply any templates defined in the script to the children of the element being processed using an `xsl:apply-templates` element as one of the rules in a template. `xsl:apply-templates` instructs the XSLT processor to treat the current element as a root element and run the templates in the script against it. For example you could rewrite [Example 147](#) as shown in [Example 148](#) using `xsl:apply-templates` and defining a template for the `title` and `subTitle` elements.

Example 148: *XSLT Template Using apply-templates*

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

Example 148: XSLT Template Using *apply-templates*

```

<xsl"template match="subTitle">
  <h2>
    <xsl:value-of select="."/>
  </h2>
</xsl:template>
</xsl:transform>

```

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

Example

For example, if your ordering system produced bills that looked similar to the XML document in [Example 149](#), you could use an XSLT script to reformat the bill for a system that required the customer's name in a single element, name, and the city and state to be in a comma-separated field, city.

Example 149: Bill XML Document

```

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

```

Example 149:*Bill XML Document*

```
<amtDue>
  123.50
</amtDue>
</widgetBill>
```

The XSLT script shown in [Example 150](#) would result in the desired transformation.

Example 150:*XSLT Script for widgetBill*

```
<xsl:transform version = '1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform' >
1  <xsl:template match="widgetBill">
  <xsl:element name="widgetBill">
    <xsl:apply-templates/>
  </xsl:element>
  </xsl:template>
2  <xsl:template match="customer">
  <xsl:element name="name">
    <xsl:value-of select="concat(//firstName,' ',//lastName)"/>
  </xsl:element>
  </xsl:template>
3  <xsl:template match="address">
  <xsl:element name="address">
    <xsl:copy-of select="//street"/>
    <xsl:element name="city">
      <xsl:value-of select="concat(//city,' ',//state)"/>
    </xsl:element>
    <xsl:copy-of select="//zipCode"/>
  </xsl:element>
  </xsl:template>
4  <xsl:template match="amtDue">
  <xsl:copy-of select="."/>
  </xsl:template>
</xsl:transform>
```

The script does the following:

1. Creates an element, `widgetBill`, in the output document and places the results of the other templates as its children.
2. Creates an element, `name`, and sets its value to the result of the concatenation.

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

Processing the document in [Example 149 on page 314](#) with this XSLT script would result in the XML document shown in [Example 151](#).

Example 151: *Processed Bill XML Document*

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

Common XSLT Functions

Overview

XSLT provides a range of capabilities in processing XML documents. These include conditional statements, looping, creating variables, and sorting. However, there are a few common functions that are used to generate output documents. These include:

- [xsl:value-of](#)
- [xsl:copy-of](#)
- [xsl:element](#)

xsl:value-of

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

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

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

xsl:copy-of

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

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

xsl:element

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

Using Codeset Conversion

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

Overview

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

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

Configuring Artix to use the i18n interceptor

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

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

For more information on configuring Artix see [Configuring and Deploying Artix Solutions](#).

Describing the codeset conversions in the contract

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

To define the codeset conversions performed by the i18n interceptor:

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

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

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

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

- ◆ `LocalCodeSet` specifies the server's native codeset. Default is the codeset specified by the local system's locale setting.

- ◆ `OutboundCodeSet` specifies the codeset into which replies are converted. Default is the codeset specified in `InboundCodeSet`.
 - ◆ `InboundCodeSet` specifies the codeset into which requests are converted. Default is the codeset specified in `LocalCodeSet`.
3. If your application includes a client that requires codeset conversion add an `i18n-context:client` element to the port definition of the service endpoint.

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

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

Example

The contract fragment in [Example 152](#) shows a port definition for an endpoint that defines a server/client pair. The server uses UTF-8 as its local codeset and the client uses ISO-8859-1 as its local codeset.

Example 152: Specifying Codeset Conversion

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

Using the endpoint definition above, the client will convert its requests into UTF-8 before sending them to the server. The server will convert its replies into ISO-8859-1 before sending them to the client. The client's

InboundCodeSet is set to ISO-8859-1 because if left unset the value would have defaulted to UTF-8. The client would then perform an extra conversion.

Index

A

- ActiveMQ 216
- Address specification
 - CORBA 253
 - HTTP 162
 - IIOp 181
 - SOAP 162
- attribute-based routing 260
 - defining rules 271

B

- bindings
 - CORBA 248
 - fixed record length 70
 - FML field tables 62
 - G2++ 147
 - SOAP with Attachments 57
 - tagged 85
 - TibrvMsg 101
 - XML 141
- bus-security 272

C

- complex types
 - all type 17
 - TibrvMsg mapping 110
 - choice type 17
 - TibrvMsg mapping 110
 - elements 17
 - occurrence constraints 18
 - sequence type 17
 - TibrvMsg mapping 109
- configuring IIOp 182
- content-based routing 260
- corba:address 253
- corba:binding 248
 - bases 248
 - repositoryID 248
- corba:corba_input_attributes 271
- corba:operation 248
 - name 248
- corba:param 248

- idltype 249
- mode 249
- name 249
- corba:policy 254
 - persistent 254
 - poaname 254
 - serviceid 254
- corba:raises 249
 - exception 249
- corba:return 249
 - idltype 249
 - name 249
- corba:typemap 246

D

- durable subscriptions 211

F

- FilenameFactoryPropertyMetaData 241
 - name 241
 - readOnly 241
 - valueSet 241
- fixed:binding 71
 - encoding 71
 - justification 71
 - padHexCode 71
- fixed:body 71
 - encoding 72
 - justification 72
 - padHexCode 72
- fixed:enumeration 75
 - fixedValue 75
 - value 75
- fixed:field 73
 - bindingOnly 72
 - fixedValue 75
 - format 73
 - size 73
- fixed:operation 71
 - discriminator 71
- fixed:sequence 79
 - counterName 80

- occurs 80
- ftp:port 228
 - connectMode 229
 - host 228
 - port 228
 - replyLocation 229
 - requestLocation 228
 - scanInterval 229
- ftp:properties 229, 237, 239
- ftp:property 229, 232, 236, 239
 - name 229, 239
 - value 229, 239
- FTPProperties 240
 - getExtensors() 240
- FTPProperty 240
- FTP Transport
 - client filename factory 231
 - reply lifecycle policy 233
 - request lifecycle policy 237
 - server filename factory 235

H

- http:address 162
 - location 162
- http-conf:client 169, 172
 - CacheControl 179
 - Conneciton 174
 - RecieveTimeout 170
 - SendTimout 170
 - UserName 172
- http-conf:HTTPServerIncomingContexts 271
- http-conf:server 169
 - CacheControl 177
 - HonorKeepAlive 174
 - RecieveTimeout 170
 - SendTimout 170

I

- i18n-context:client 320
 - InboundCodeSet 321
 - LocalCodeSet 321
 - OutboundCodeSet 321
- i18n-context:server 320
 - InboundCodeSet 321
 - LocalCodeSet 320
 - OutboundCodeSet 321
- ignorecase 272
- iiop:address 181

- iiop:payload 182
- iiop:policy 182
 - persistent 183
 - poaname 183
 - serviceid 183
- IOR specification 181, 253

J

- Java Messaging System 204
- Java Naming and Directory Interface 204
- JMS 204
 - jms:address 206
 - connectionPassword attribute 206
 - connectionUserName attribute 206
 - destinationStyle attribute 206
 - durableSubscriberName 211
 - jndiConnectionFactoryName attribute 206
 - jndiDestinationName attribute 206
 - jndiReplyDestinationName attribute 206
 - messageSelector 211
 - transactional 212
 - jms:client 210
 - messageType attribute 210
 - jms:JMSNamingProperties
 - name attribute 207
 - value attribute 207
 - jms:JMSNamingProperty 207
 - jms:server 211
 - durableSubscriberName attribute 211
 - messageSelector attribute 211
 - transactional attribute 211
 - useMessageIDAsCorrealationID attribute 211
- JNDI 204

M

- mime:content 57
 - part 57
 - type 57
- mime:multipartRelated 56
- mime:part 56, 57
 - name 57
- mq:client 188
 - AliasQueueName 196
 - Delievery 198
 - Format 200
 - Server_Client 194
 - Transactional 198
- mq:IncomingMessageAttributes 271

- mq:server 188
 - Delivery 198
 - Format 200
 - Server_Client 194
 - Transactional 198

N

- NMTOKEN
 - TibrvMsg mapping 111

O

- operation-based routing 260, 263
 - defining rules 268

P

- plugins:ftp:policy:client:filenameFactory 233
- plugins:ftp:policy:client:replyFileLifecycle 233
- plugins:ftp:policy:server:filenameFactory 238
- plugins:ftp:policy:server:requestFileLifecycle 238
- port-based routing 260, 261
 - defining rules 265

R

- routing
 - broadcast 284
 - failover 286
 - fanout 284
 - load balancing 283
- routing:contains 272
- routing:desitnation 277
- routing:destination 266, 288
 - id attribute 277
 - port 266
 - route 288
 - service 266
- routing:empty 272
- routing:endswith 272
- routing:equals 272
 - contextAttributeName 271
 - contextName 271
 - value 272
- routing:expression 276
 - evaluator attribute 276
 - name attribute 276
- routing:greater 272
- routing:less 272
- routing:nonempty 272

- routing:operation 268
 - name 268
 - target 268
- routing:query 277
 - expression attribute 277
- routing:route 265
 - multiRoute 283, 284, 286
 - failover 286
 - fanout 284
 - loadBalance 283
 - name 265
- routing:source 265
 - port 265
 - service 265
- routing:startswith 272
- routing:transportAttribute 271

S

- soap:address 162
 - location 162
- soap:body
 - parts 52
- soap:header 51
 - encodingStyle 51
 - message 51
 - namespace 51
 - part 51
 - use 51
- Specifying POA policies 182, 254

T

- tagged:binding 86
- tagged:body 88
- tagged:case 92
- tagged:choice 92
- tagged:enumeration 89
- tagged:field 89
- tagged:operation 88
- tagged:sequence 90
- tibrv:array 113
- tibrv:binding 102
 - stringAsOpaque 102
 - stringEncoding 102
- tibrv:context 137
- tibrv:field 135
- tibrv:input 103
- tibrv:msg 135
- tibrv:operation 103

- tibrv:output 104
 - messageNameFieldPath 104
 - messageNameFieldValue 105
 - stringAsOpaque 105
 - stringEncoding 105
- tibrv:port 218
 - serverSubject 218
- tuxedo:binding 67
- tuxedo:field 67
 - id 67
 - name 67
- tuxedo:fieldTable 67
- tuxedo:input 223
 - operation 223
- tuxedo:operation 67
- tuxedo:server 223
- tuxedo:service 223
 - name 223

W

- WebSphere MQ
 - Format
 - working with mainframes 201
- WSDL 1
 - binding element 2, 44
 - name attribute 44
 - definitions element 2, 56
 - message element 2, 29
 - operation element 2
 - part element 31
 - element attribute 31
 - name attribute 31
 - type attribute 31
 - port element 3, 158
 - binding attribute 158
 - portType element 2, 34
 - schema element 11
 - targetNamespace attribute 11
 - service element 2, 158
 - name attribute 158
 - types element 2, 11
- wsdltoCorba 247, 257
- wsdltoService
 - adding a CORBA service 255
 - adding a JMS service 213
 - adding an HTTP service 163
 - adding an IIOP service 183
 - adding a TIBCO service 218
 - adding a Tuxedo service 224

- adding a WebSphere MQ service 189
- wsdltoSoap 48

X

- xformat:binding 141
 - rootNode 142
- xformat:body 142
 - rootNode 142
- XMLSchema 9
 - all element 17
 - choice element 17
 - complexType element 16
 - element element 17
 - maxOccurs attribute 18
 - minOccurs attribute 18
 - name attribute 18
 - type attribute 18
 - sequence element 17
- XML Stylesheet Language Transformations 308
- XPath 311
- xsl:apply-templates 313
 - select 314
- xsl:copy-of 317
 - select 317
- xsl:element 317
 - name 317
 - namespace 317
- xsl:stylesheet 309
- xsl:template 311
 - match 311
- xsl:transform 309
- xsl:value-of 317
 - select 317
- XSLT 308

