Artix™

Designing Artix Solutions

Version 3.0, October 2005

Making Software Work Together™

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.

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 Technologies PLC 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.

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 © 1999-2005 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

Contents

List of Figures	**xi**
List of Tables	**xv**
Preface	**xvii**
What is Covered in this Book	xvii
Who Should Read this Book	xvii
How to Use this Book	xvii
Finding Your Way Around the Library	xviii
Searching the Artix Library	xx
Online Help	xx
Additional Resources	xx
Document Conventions	xxi
Chapter 1 Introducing Artix	**1**
What is Artix?	2
Artix Contracts and WSDL	3
Beyond the Contract	6
Chapter 2 Getting Started with Artix Designer	**7**
Introducing Artix Designer	8
Setting Up Artix Designer	9
Starting Artix Designer	11
Setting Artix Designer Preferences	13
Chapter 3 Creating an Artix Designer Project	**17**
What is an Artix Designer Project?	18
Creating a Project	20
Creating a Basic Web Services Project	21
Creating a CORBA Web Services Project	23
Creating a Project Using a Template	27

Chapter 4 Creating Artix Resources 31

What are Artix Resources? 32

Creating Design Resources 34

Creating a New Contract 35

Importing a Contract from a URL 38

Creating a Contract from CORBA IDL 40

Creating a Contract from a Java Class 50

Creating a Contract from a COBOL Copybook 62

Creating a Contract from a Data Set 69

Creating a Contract from an XML Schema Document 78

Creating an XML Schema 80

Importing an XML Schema from a URL 82

Creating Access Control Lists 83

Working with Generation Profiles 85

Chapter 5 Defining Data Types 91

Introducing Data Types 92

Creating New Type Systems 93

Specifying a Type System in a Contract 95

XML Schema Simple Types 96

Defining Complex Data Types 98

Defining Data Structures 99

Defining Arrays 107

Defining Types by Extension 109

Defining Types by Restriction 115

Defining Enumerated Types 117

Defining Elements 120

Chapter 6 Defining Messages 123

Chapter 7 Defining Your Interfaces 127

Chapter 8 Binding Interfaces to a Payload Format 135

Introducing Bindings 136

Adding a SOAP Binding 137

Adding a Default SOAP Binding 138

Adding SOAP Headers to a SOAP Binding 142

Sending Data Using SOAP with Attachments 148
Adding a CORBA Binding 152
Adding an FML Binding 159
Adding a Fixed Binding 166
Adding a Tagged Binding 186
Adding a TibrvMsg Binding 203
Defining a TibrvMsg Binding 204
Defining Array Mapping Policies 211
Defining a Custom TibrvMsg Mapping 216
Adding Context Information to a TibrvMsg 234
Adding a Pure XML Binding 237
Adding a G2++ Binding 243

Chapter 9 Adding Transports 251
Introducing Services 252
Defining a Service 253
Creating an HTTP Service 255
Creating a CORBA Service 264
Configuring an Artix CORBA Port 265
Generating CORBA IDL 271
Creating an IIOP Service 273
Creating a WebSphere MQ Service 279
Creating a Java Messaging System Service 288
Adding a TIBCO Service 298
Creating a Tuxedo Service 305
Configuring a Service to Use Codeset Conversion 309

Chapter 10 Adding Routing Instructions 313
Artix Routing 314
Compatibility of Ports and Operations 315
Defining Routes in Artix Contracts 318
Using Port-Based Routing 319
Using Operation-Based Routing 322
Advanced Routing Features 325
Creating Routes Using Artix Designer 330
Creating Routes from the Command Line 334
Load Balancing 338
Error Handling 339

Service Lifecycles 340
Routing References to Transient Servants 342

Chapter 11 Fastrack Service Enabling 345
Web Service Enabling a Service 346
CORBA Enabling a Service 348

Chapter 12 Editing Artix Resources 351
Editing Contracts and Schemas 352
Working with the Editor Views 353
Editing Types 356
Editing Messages 357
Editing Port Types 359
Editing Bindings 363
Editing Services 364
Editing Routes 365
Editing Generated Resources 367

Chapter 13 Using the Artix Transformer 369
Using the Artix Transformer as an Artix Server 370
Using Artix to Facilitate Interface Versioning 372
WSDL Messages and the Transformer 377
Writing XSLT Scripts 380
Elements of an XSLT Script 381
XSLT Templates 383
Common XSLT Functions 389

Appendix A SOAP Binding Extensions 391
soap:binding element 392
soap:operation element 394
soap:body element 395
soap:header element 398
soap:fault element 400
soap:address element 402

Appendix B CORBA Type Mapping 403
Introducing CORBA Type Mapping 404

Primitive Type Mapping **405**
Complex Type Mapping **408**
Structures 409
Enumerations 411
Fixed 412
Unions 414
Type Renaming 417
Arrays 418
Multidimensional Arrays 420
Sequences 421
Exceptions 423
Recursive Type Mapping **425**
Mapping XML Schema Features that are not Native to IDL **427**
Binary Types 428
Attributes 429
Nested Choices 431
Inheritance 433
Nillable 436
Optional Attributes 438
Artix References **440**

Appendix C TibrvMsg Default Mappings **445**

Appendix D HTTP Port Properties **451**
Defining an HTTP Port **452**
HTTP Client Configuration **453**
HTTP Server Configuration **456**
HTTP Attribute Details **459**
SendTimeout 460
ReceiveTimeout 461
AutoRedirect 462
UserName 463
Password 464
AuthorizationType 465
Authorization 466
Accept 467
AcceptLanguage 468
AcceptEncoding 469

ContentType 470
ContentEncoding 471
Host 472
Connection 473
CacheControl 474
Cookie 478
BrowserType 479
Referer 480
ProxyServer 481
ProxyAuthorizationType 482
ProxyAuthorization 483
UseSecureSockets 484
ClientPrivateKey 485
SuppressClientSendErrors 486
SuppressClientReceiveErrors 487
HonorKeepAlive 488
RedirectURL 489
ServerType 490
ServerCertificateChain 491

Appendix E WebSphere MQ Port Properties 493

Defining an MQ Port 494

MQ Port Attributes 497

QueueManager 498
QueueName 499
ReplyQueueName 500
ReplyQueueManager 501
Server_Client 502
ModelQueueName 503
AliasQueueName 504
ConnectionName 506
ConnectionReusable 507
ConnectionFastPath 508
UsageStyle 509
CorrelationStyle 510
AccessMode 512
Timeout 514
MessageExpiry 515
MessagePriority 516

Delivery 517
Transactional 518
ReportOption 520
Format 522
MessageId 524
CorrelationId 525
ApplicationData 526
AccountingToken 527
Convert 528
ApplicationIdData 529
ApplicationOriginData 530
UserIdentification 531

Appendix F Tibco Port Properties 533

Glossary 541

Index 547

List of Figures

Figure 1: Workspace Selection 11
Figure 2: The Welcome Screen 12
Figure 3: The Artix Designer Icon 12
Figure 4: Eclipse Preferences Window 13
Figure 5: Common Artix Designer Preferences 14
Figure 6: Artix Designer Java Preferences 14
Figure 7: Artix Designer C++ Preferences 15
Figure 8: Artix Designer Validation Preferences 16
Figure 9: Project Structure 19
Figure 10: New Project Wizard 21
Figure 11: Basic Web Services Project Details 22
Figure 12: CORBA Web Services Project Details 23
Figure 13: CORBA Web Services Project Input Details 24
Figure 14: The Naming Service Contexts Dialog Box 25
Figure 15: The New Window 35
Figure 16: Entering WSDL Details 36
Figure 17: Entering WSDL Link Details 37
Figure 18: Entering WSDL URL Details 38
Figure 19: Entering Contract Name 41
Figure 20: Specifying CORBA IDL Details 42
Figure 21: Entering Contract Name for Java Import 51
Figure 22: Java Class Details 52
Figure 23: The Find Class Dialog Box 53
Figure 24: Select Data Format 62
Figure 25: The Set Defaults Window 63
Figure 26: The Operations Editor 64

Figure 27: Editing an Operation 65
Figure 28: Editing a Message 66
Figure 29: Editing a Fixed Field 71
Figure 30: Entering Defaults for a Contract from a Tagged Data Set 73
Figure 31: Editing a Tagged Data Set Operation 75
Figure 32: Editing a Tagged Data Set Message 76
Figure 33: Selecting Roles for an Access Control List 83
Figure 34: Editing ACL Roles for Operations 84
Figure 35: The Artix Generator Window 85
Figure 36: The Artix Generator General Tabbed page 86
Figure 37: The Artix Generator Generation Tabbed Page 87
Figure 38: The Artix Generator WSDL Details Tabbed Page 88
Figure 39: Select Resource to which a New Type is Added 100
Figure 40: Defining a Type's General Properties 101
Figure 41: Adding Elements to a Structure 102
Figure 42: Adding Attributes to a Structure 103
Figure 43: Defining the Base Type for Extension 111
Figure 44: Adding Elements to an Extended Type 112
Figure 45: Adding Global Attributes to an Extended Complex Type 113
Figure 46: Defining the Values for an Enumeration 118
Figure 47: Defining the Base Values for an Element Definition 121
Figure 48: Naming a Message 125
Figure 49: Adding Parts to a Message 126
Figure 50: Naming a Port Type 130
Figure 51: Adding an Operation to a New Port Type 131
Figure 52: Defining the Messages in an Operation 132
Figure 53: Select the Type of Binding to Use 138
Figure 54: Setting the Defaults for a SOAP Binding 139
Figure 55: Setting CORBA Binding Defaults 153

Figure 56: Setting the Default Values for a Fixed Binding 167
Figure 57: Editing the Fixed Binding Settings 168
Figure 58: Editing an Operation's Fixed Binding Settings 168
Figure 59: Editing a Message's Fixed Binding Settings 169
Figure 60: Setting the Default Values for a Tagged Binding 187
Figure 61: Editing the Operations in the Fixed Binding 188
Figure 62: Editing the Tagged Binding Settings for an Operation 189
Figure 63: Editing a Message's Tagged Binding Settings 190
Figure 64: Setting the Default Values for an XML Binding 238
Figure 65: Specifying the Name and Binding for a Port 256
Figure 66: Setting the HTTP Transport Attributes 257
Figure 67: Specifying the Address of a CORBA Endpoint 268
Figure 68: Editing IIOP Tunnel Transport Attributes 274
Figure 69: Editing WebSphere MQ Transport Attributes 281
Figure 70: Editing JMS Transport Attributes 293
Figure 71: Editing TIBCO Transport Attributes 300
Figure 72: Editing Tuxedo Transport Attributes 306
Figure 73: Defining the Endpoints of a Route 331
Figure 74: Selecting the Operations to Use for the Route 332
Figure 75: Specifying Transport Attributes to Constrain a Route 333
Figure 76: SOAP Binding and Service Details Window 346
Figure 77: CORBA Binding and Service Details Window 348
Figure 78: Editing in Diagram View 353
Figure 79: Editing in Source View 354
Figure 80: The Outline View 355
Figure 81: MQ Remote Queues 505

List of Tables

Table 1: Artix Code Generation Profiles 32
Table 2: Java to WSDL Mappings 55
Table 3: complexType Descriptor Elements 104
Table 4: Part Data Type Attributes 124
Table 5: Operation Message Elements 128
Table 6: Attributes of the Input and Output Elements 129
Table 7: FML Type Support 159
Table 8: Attributes for fixed:binding 171
Table 9: Attributes for tagged:binding 192
Table 10: Attributes for tagged:operation 193
Table 11: Attributes for tagged:field 194
Table 12: Attributes for tagged:sequence 195
Table 13: Attributes for tagged:choice 197
Table 14: Attributes for tibrv:binding 205
Table 15: Attributes for tibrv:input 206
Table 16: Attributes for tibrv:output 207
Table 17: Effect of tibrv:array 211
Table 18: Attributes for tibrv:array 212
Table 19: Functions Used for Specifying TibrvMsg Array Element Names 213
Table 20: Valid Casts for TibrvMsg Binding 219
Table 21: Attributes for tibrv:msg 231
Table 22: Attributes for tibrv:field 232
Table 23: Required JMS Port Attributes 289
Table 24: Supported TIBCO Rendezvous Features 298
Table 25: Context QNames 328
Table 26: Context Names Used with wsdltorouting 335

Table 27: Conditions Used with wsdltorouting 336
Table 28: Attributes for soap:header 399
Table 29: soap:fault attributes 401
Table 30: Primitive Type Mapping for CORBA Plug-in 405
Table 31: Complex Type Mapping for CORBA Plug-in 408
Table 32: Complex Content Identifiers in CORBA Typemap 433
Table 33: TIBCO to XSD Type Mapping 445
Table 34: HTTP Client Configuration Attributes 453
Table 35: HTTP Server Configuration Attributes 456
Table 36: Settings for http-conf:client CacheControl 474
Table 37: Settings for http-conf:server CacheControl 475
Table 38: WebSphere MQ Port Attributes 494
Table 39: Server_Client Settings for the MQ Transport 502
Table 40: UsageStyle Settings 509
Table 41: MQGET and MQPUT Actions 510
Table 42: Artix WebSphere MQ Access Modes 512
Table 43: Transactional Attribute Settings 518
Table 44: ReportOption Attribute Settings 520
Table 45: FormatType Attribute Settings 522
Table 46: TIB/RV Transport Properties 533
Table 47: TIB/RV Supported Payload formats 536

Preface

What is Covered in this Book

Designing Artix Solutions outlines how to design, develop, and deploy integration solutions with Artix using the graphical user interface (GUI), the Artix command line tools, or both. It also guides you through producing Web Services Description Language (WSDL), source code, and runtime configuration files for your Artix integration solution.

For information on using the transports discussed in the book, such as WebSphere MQ or Tuxedo, refer to the documentation for that product.

Who Should Read this Book

This guide is intended for all users of Artix. This guide assumes that you have a working knowledge of the middleware transports that are being used to implement the Artix system. It also assumes that you are familiar with basic software design concepts, and that you have a basic understanding of WSDL.

How to Use this Book

If you are new to Artix

To learn about Artix, see "Introducing Artix" on page 1.

If you encounter terms you are unfamiliar with, turn to the "Glossary" on page 541 for a list of Artix terms and definitions.

If you have used Artix before

You probably know what you want to do with Artix. In this case, one of the following suggestions may help.

- If you are creating a new project, see "Creating an Artix Designer Project"

- If you're creating Artix resources, see "Creating Artix Resources"
- If you need to create or edit the data types, see "Defining Data Types"
- If you need to add a message or edit the existing messages used by your system, see "Defining Messages"
- If you need to create or edit an interface definition, see "Defining Your Interfaces"
- If you're creating or editing a binding, see "Binding Interfaces to a Payload Format"
- If you're creating or editing a service, see "Adding Transports"
- If you're creating or editing a route, see "Adding Routing Instructions"
- If you are editing Artix resources using the Artix designer, see "Editing Artix Resources".

In addition, the following appendices are included to provide reference material on using some the Artix bindings:

- "SOAP Binding Extensions" on page 391
- "CORBA Type Mapping" on page 403
- "WebSphere MQ Port Properties" on page 493
- "Tibco Port Properties" on page 533

Finding Your Way Around the Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The Artix library is listed here, with a short description of each book.

If you are new to Artix

You may be interested in reading:

- 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.

To design and develop Artix solutions

Read one or more of the following:

- Designing Artix Solutions provides detailed information about describing services in Artix contracts and using Artix services to solve problems.

- Developing Artix Applications in C++ discusses the technical aspects of programming applications using the C++ API.
- Developing Artix Plug-ins with C++ discusses the technical aspects of implementing plug-ins to the Artix bus using the C++ API.
- Developing Artix Applications in Java discusses the technical aspects of programming applications using the Java API.
- Artix for CORBA provides detailed information on using Artix in a CORBA environment.
- Artix for J2EE provides detailed information on using Artix to integrate with J2EE applications.
- Artix Technical Use Cases provides a number of step-by-step examples of building common Artix solutions.

To configure and manage your Artix solution

Read one or more of the following:

- Deploying and Managing Artix Solutions describes how to deploy Artix-enabled systems, and provides detailed examples for a number of typical use cases.
- Artix Configuration Guide explains how to configure your Artix environment. It also provides reference information on Artix configuration variables.
- IONA Tivoli Integration Guide explains how to integrate Artix with IBM Tivoli.
- IONA BMC Patrol Integration Guide explains how to integrate Artix with BMC Patrol.
- Artix Security Guide provides detailed information about using the security features of Artix.

Reference material

In addition to the technical guides, the Artix library includes the following reference manuals:

- Artix Command Line Reference
- Artix C++ API Reference
- Artix Java API Reference

Have you got 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/3.0/index.xml

You can also search within a particular book. To search within an 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.

Online Help

Artix Designer includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index, and glossary.
- A full search feature.
- Context-sensitive help.

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

- Click the Help button on the Artix Designer panel, or
- Select **Contents** from the Help menu

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:

`Fixed width`	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 `IT_Bus::AnyType` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
`Fixed width italic`	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: `% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
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**).

CHAPTER 1

Introducing Artix

Artix gives you the tools to design, develop, and deploy service oriented integration solutions.

In this chapter

This chapter discusses the following topics:

What is Artix?	page 2
Artix Contracts and WSDL	page 3
Beyond the Contract	page 6

What is Artix?

Overview

Artix is a flexible and easy-to-use tool for integrating your existing applications across a number of different middleware platforms. Artix also makes it easy to expose your existing applications as services for any number of applications using other middleware transports.

In addition, Artix provides a flexible programming model that allows you to create new services that can communicate using any of the protocols that Artix supports.

Despite the flexibility and power of Artix, designing solutions using Artix is a straightforward process.

Artix Contracts and WSDL

Overview

When designing Artix solutions, 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 the root `definition` element. These elements describe a service and how that service can is 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 complex data types. 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 operations provided 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 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 specfiy the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced by the WSDL.

- `types`—the definition of complex data types. This element contains XML Schema definitions of any complex datatypes used by your service. For information about defining datatypes see "Defining Data Types" on page 91.
- `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 Messages" on page 123.
- `portType`—a collection of `operation` elements representing an abstract endpoint. The port type elements can be thought of as the interface description for a service. In fact, Artix generates code using a one-to-one relationship between a port type and an interface. For inforamtion about defining port types see "Defining Your Interfaces" on page 127.
- `operation`—the abstract description of an action perfromed by a service. For information on defining operaitons see "Defining Your Interfaces" on page 127.
- `binding`—the concrete data format specification for a port type. A binding element defines how the abstract messages are mapped into the concrete data format sent between endpoints. This is where specifics such as parameter order and return values are specified. For information on defining bindings see "Binding Interfaces to a Payload Format" on page 135.
- `service`—a collection of related `port` elements. These elements are respositories for organizing endpoint definitions.
- `port`—the endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combine it with the definition of transport details, and define the physical endpoint on which a service is exposed. For information on defining endpoints see "Adding Transports" on page 251.

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 solution you must perform the following steps:

1. Define the data types used in your solution.
2. Define the messages used in your solution.
3. Define the interfaces for each of the services in your solution.
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 in your solution.
6. Define any routing rules used in your solution.

Beyond the Contract

Overview

After you have created the contract defining your Artix solution, you still have work to do before your solution is ready to go. There are two remaining steps in developing a solution using Artix:

1. Develop any application-level code needed to complete the solution.
2. Configure the Artix components.

Developing application code

Often, you will need to develop new application logic as a part of your solution. Artix provides tools that allow you to develop this new functionality using familiar programming paradigms.

For example, if you are a Java developer writing a client to interact with a Tuxedo application, Artix will generate the Java stub code needed to develop the application using either pure Java or as a J2EE bean.

Artix provides code generators to create stub and skeleton code in C++ and Java. The APIs used by Artix make it easy to develop transport-independent, Web services-based applications using standard programming techniques. For more information on developing Artix applications, see Developing Artix Applications in C++ or Developing Artix Applications in Java.

In addition, Artix provides tools for generating CORBA IDL from an Artix contract. For more information, see Artix for CORBA.

Configuring the Artix components

Before deploying your Artix solution you need to configure the runtime environment for your Artix components and services. For a detailed discussion of Artix configuration, see the Deploying and Managing Artix Solutions.

Getting Started with Artix Designer

Artix Designer is a comprehensive service development environment that ships as a plug-in to the Eclipse platform.

In this chapter

This chapter discusses the following topics:

Introducing Artix Designer	page 8
Setting Up Artix Designer	page 9
Starting Artix Designer	page 11
Setting Artix Designer Preferences	page 13

Introducing Artix Designer

Overview

Artix Designer is an Eclipse-based development environment that provides a unified workspace for building Artix contracts, writing application code, configuring Artix, and debugging solutions. By adding Artix contract creation wizards and Artix code generators to the Eclipse platform, Artix Designer provides many of the tools needed to build an Artix project from scratch.

Setting Up Artix Designer

Overview

When you install Artix, you get a default installation of Artix Designer. This includes the

- Eclipse platform
- Eclipse Java Development Tools (JDT)
- Eclipse C/C++ Development Tools (CDT)
- Eclipse EMF plug-ins
- Artix Designer plug-ins

In addition, the installer places a start script for Artix Designer into your installation's `bin` directory.

With the default installation Artix Designer provides the following features:

- Wizards for adding XML Schema type definitions to your contracts
- Wizards for defining WSDL messages
- Wizards for Defining WSDL portType definitions for your service interfaces
- Wizards for binding your service interfaces to a number of the payload formats supported by Artix
- Wizards for adding the WSDL service elements needed to expose your services using Artix transports
- Wizards for importing a CORBA IDL file into an Artix project and building contracts based on it
- Wizards for importing a COBOL copybook into an Artix project and building contracts based on it
- A Java code generator
- A C++ code generator
- An IDL generator
- Tools for configuring Artix services
- A Java compiler and debugger
- A WSDL editor with built-in validation

Using with an existing Eclipse installation

If you have an existing Eclipse installation, you can plug Artix Designer into it. This way you can retain any of the plug-ins and setting you already use. To plug Artix Designer into an existing Eclipse installation extract *InstallDir*/artix/3.0/eclipse/ArtixDesignerPlugin.zip to the root folder of your Eclipse installation.

When you start Eclipse, you must use the start_eclispe script if you want to use the Artix Designer features.

Starting Artix Designer

Windows

On Windows platforms, you start Artix Designer using the **Start** menu shortcut. If you installed Artix using the default settings the short cut for stating Artix Designer is under **Start|(All) Programs|IONA|Artix 3.0|Artix Designer**.

You can also use the `start_eclipse` script located under *InstallDir*`\artix\3.0\bin\`.

UNIX

On UNIX and Linux platforms you start Artix Designer using the supplied `start_eclipse` script. The script is located in *InstallDir*`/artix/3.0/bin`.

Selecting the default workspace

When you start Artix Designer, a dialog box, shown in Figure 1, asks you to select a workspace. The workspace is the root folder into which all of your Eclipse projects are stored. You can either accept the recommended default or enter a new workspace.

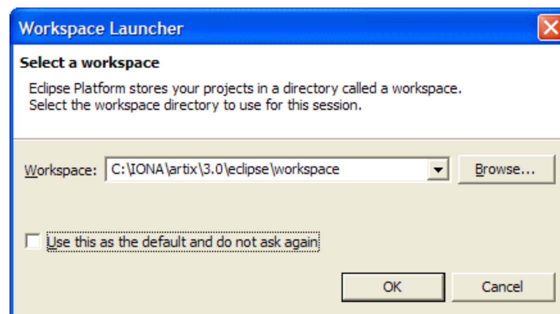


Figure 1: *Workspace Selection*

If you do not want to see this dialog at start-up, select the **Use this as the default and do not ask again** check box. Artix Designer will open in the specified workspace from now on.

Once Artix Designer is running, you can change workspaces by selecting **File|Switch Workspace.**

The Welcome screen

When you launch Artix Designer for the first time, the Welcome screen will display.

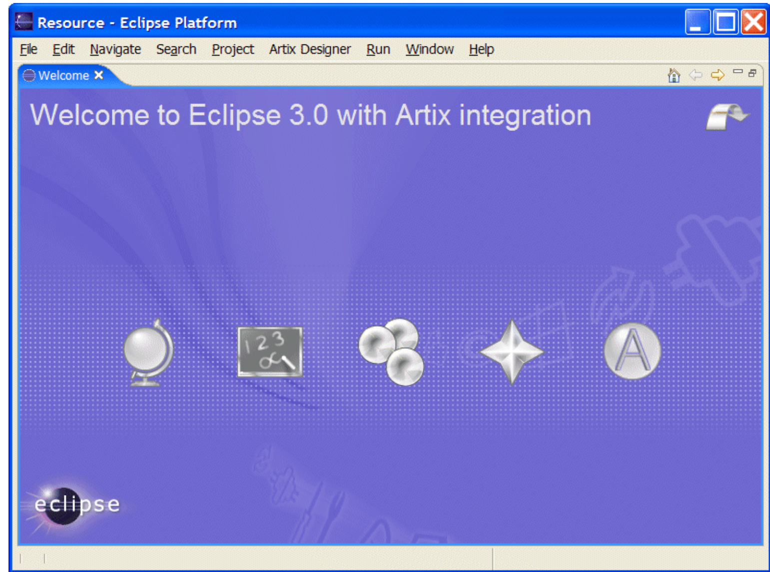


Figure 2: *The Welcome Screen*

You can click the **Artix Designer** icon on the right to access links to a series of tutorials, release notes, online support, and the Artix documentation set.

Figure 3: *The Artix Designer Icon*

Setting Artix Designer Preferences

Overview

Artix Designer uses a number of preferences that determine the defaults used by the project templates. These preferences also specify the location of your Artix installation and the Artix environment script used by Artix Designer.

Opening the preference window

The Artix preferences are set through the same window as the Eclipse preferences. To open the Eclipse preference widow select **Window|Preferences** from the toolbar. The Preferences window will open, as shown in Figure 4.

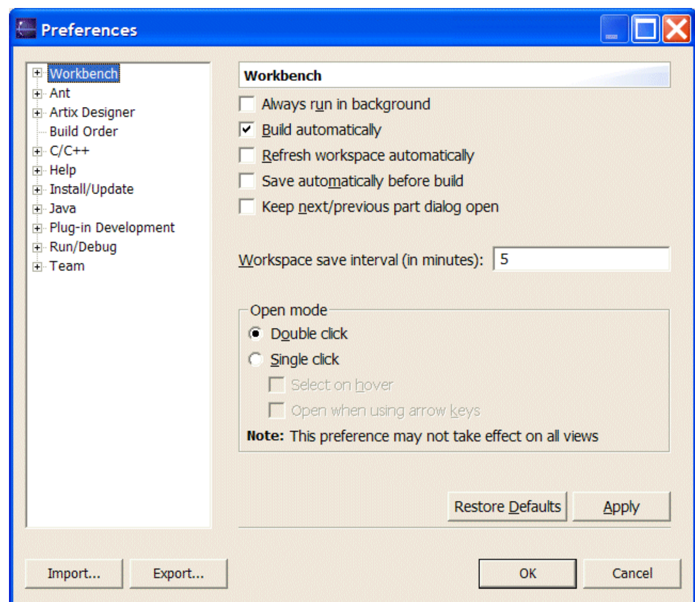


Figure 4: *Eclipse Preferences Window*

The Artix preferences are located under the **Artix Designer** entry.

Common preferences

Artix Designer needs to be pointed to an Artix installation in order to work properly. The settings for where Artix Designer finds its tools and environment scripts is set in the **Artix Designer|Common** panel of the **Preferences** window, as shown in Figure 5.

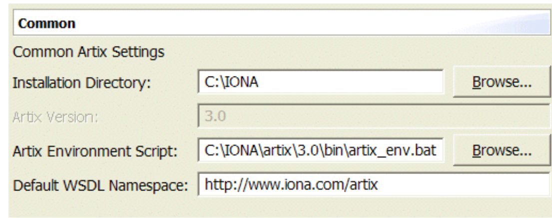


Figure 5: *Common Artix Designer Preferences*

In general, you should not need to change these settings.

Java code generation preferences

The **Artix Designer|Java** preferences panel, shown in Figure 6, allows you to set the default package name used by the Artix Java code generator when you use a template to create your projects. The default package name is also used as the default for any Artix code generation configuration that is set up to generate Java code.

Figure 6: *Artix Designer Java Preferences*

If you want to ensure that your edits are maintained if you regenerate the Artix Java stubs and Java skeletons, select **Merge generated code**.

C++ code generation preferences

The **Artix Designer|C++** preferences panel, shown in Figure 7, allows you to set the default C++ namespace and the default C++ declaration statement used by the Artix C++ code generator when you use a template to create your projects. The default settings are also used as the default for any Artix code generation configuration that is set up to generate C++ code.

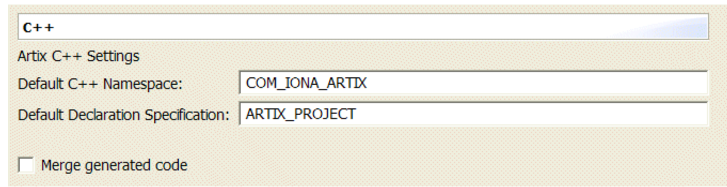


Figure 7: *Artix Designer C++ Preferences*

If you want to ensure that your edits are maintained if you regenerate the Artix C++ stubs and C++ skeletons, select **Merge generated code**.

> **Note:** If you select **Merge generated code** for C++ projects, Eclipse will launch a compare editor each time you regenerate C++ code. This enables you to choose which code changes you want to retain.

Validation preferences

The **Artix Designer|Validation** panel, shown in Figure 8 allows you to enable support for extra schema validation, by selecting the **Validate WSDL against schemas** check box. You can add schemas to the Additional Schemas list by clicking the **New** button and choosing a schema from your file system.

Where you have multiple additional schemas, you can reorder the validation sequence using the **Up** and **Down** buttons.

Select the **Check for references prior to deletion** check box to ensure that you will be warned if you try to delete an element that is referenced elsewhere, thus invalidating your WSDL.

You can choose whether Artix Designer restricts reference-checking to the same file, the same project, or your entire workspace by selecting the appropriate radio button in the Scope section.

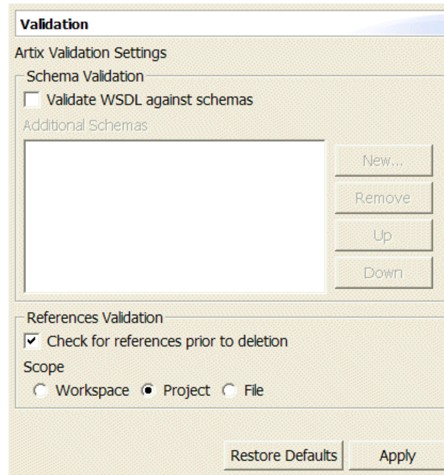


Figure 8: *Artix Designer Validation Preferences*

Creating an Artix Designer Project

Artix projects provide a workspace within which you can design Artix solutions.

In this chapter

This chapter discusses the following topics:

What is an Artix Designer Project?	page 18
Creating a Project	page 20
Creating a Project Using a Template	page 27

What is an Artix Designer Project?

Overview

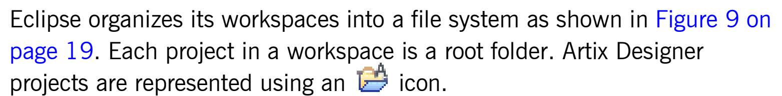
An Artix Designer project is an Eclipse project that uses the Artix perspective and includes the resources needed to build and deploy an Artix solution. The resources included in an Artix Designer project include:

- Artix contracts
- XML Schema documents
- Generated Java code
- Generated C++ code
- Artix configuration files

You can also include other resources such as COBOL copybooks, text files, scripts, or any other resources you want to group with a particular project.

In general, an Artix Designer project is intended to encompass a single Artix solution. This, however, does not mean that a project cannot include a number of services or applications. Often a single Artix solution has many parts to it and all of these parts can associated with a common Artix project.

Organization of an Artix Project

Eclipse organizes its workspaces into a file system as shown in Figure 9 on page 19. Each project in a workspace is a root folder. Artix Designer projects are represented using an  icon.

At a minimum, an Artix Designer project will contain a `.project` file that Eclipse uses to store information about the project and an `outputs` folder that Artix uses to store generated resources for your project.

Projects that contain generated Java code will also contain a `.classpath` file to define the classpath used to build the code and a `bin` folder to hold the compiled Java classes.

Projects that contain generated C++ code will contain a `.cdtproject` file.

An Artix project's `outputs` folder contains a number of sub-folders. Each sub-folder groups together Artix generated resources according to the details selected when creating the resource's generation profile. For example, if you select to generate resources for an Artix switch, the generated resources

would be placed in a folder under the `switches` sub-folder. For more information on the different types of generation profiles see Table 1 on page 32.

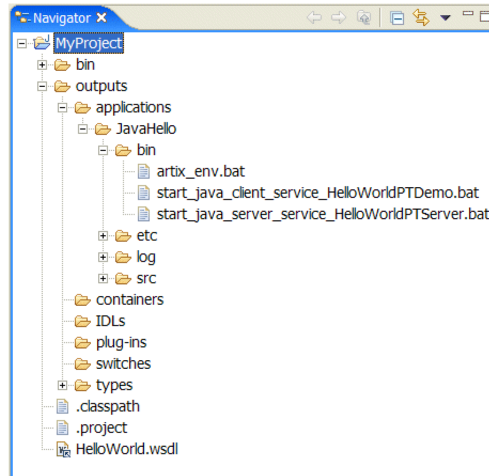


Figure 9: *Project Structure*

Types of projects

Artix Designer provides two types of project:

Basic Web services project includes most of what you might do with Artix. Select this type of project if you want to write a WSDL or XSD file or develop an application from a new or pre-existing WSDL file. One option when creating a basic Web services project is to use a code generation template that is configured to automatically create typical client, server, or switch applications.

CORBA Web services project is a special purpose project that allows rapid design of an application in which an Artix client, using SOAP over HTTP messaging, can communicate with a pre-existing CORBA server.

Creating a Project

Overview

All development done in Artix Designer is associated with a project. Artix provides wizards that guide you through the creation of an Artix Designer project.

In this section

This section discusses the following topics:

Creating a Basic Web Services Project	page 21
Creating a CORBA Web Services Project	page 23

Creating a Basic Web Services Project

Overview

A basic Web services project provides a base for all Artix projects. From this project type, you can create any type of Artix solution. It provides you with a completely blank slate from which to start developing. However, there are a number of templates that you can use to help you create starting point elements. For information on using templates see "Creating a Project Using a Template" on page 27.

Procedure

To create a basic web services:

1. Select **File|New|Project** to open the new project wizard shown in Figure 10.

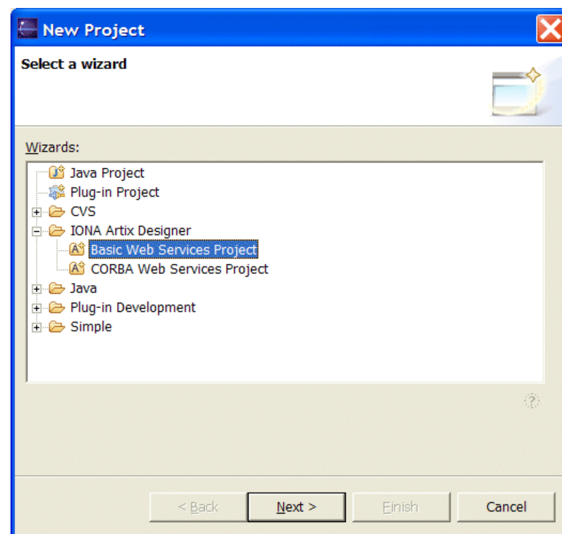


Figure 10: *New Project Wizard*

2. Select **Basic Web Services Project** under the **IONA Artix Designer** folder.

3. Click **Next** to open the **Basic Web Services Project General Details** window, shown in Figure 11.

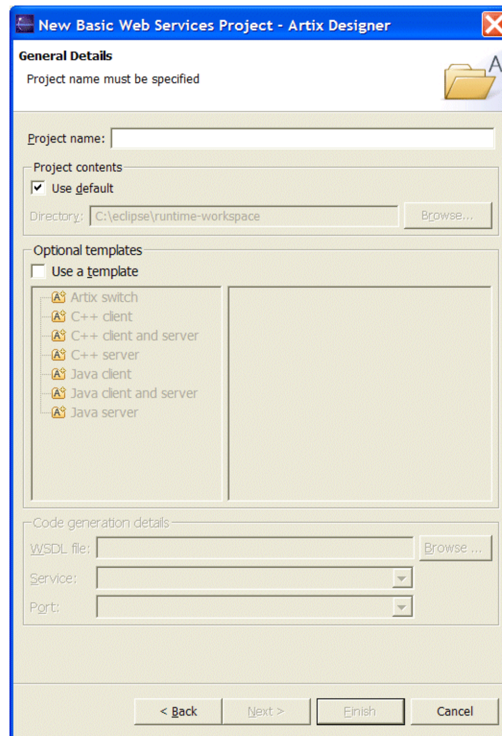


Figure 11: *Basic Web Services Project Details*

4. Enter a project name.
5. If you do not want to place your project into the default workspace being used by Eclipse, clear the **Use default** check box and enter a new directory in which to store your project.
6. If you want to use a template for your project select the **Use a template** check box. See “Creating a Project Using a Template” on page 27.
7. Click **Finish**.

Creating a CORBA Web Services Project

Overview

A CORBA Web Services Project is a special purpose project that allows rapid design of an application in which a Web service client can communicate with a pre-existing CORBA server. When you create this type of project, Artix Designer walks you through the process of importing the IDL definition of the running service, discovering the CORBA service's IOR, and identifying the endpoint to route requests from.

Procedure

To create a CORBA Web services project:

1. Select **File|New|Project** to open the new project wizard.
2. Select **CORBA Web Services Project** under the **IONA Artix Designer** folder.
3. Click **Next** to open the **CORBA Web Services Project General Details** window, shown in Figure 12.

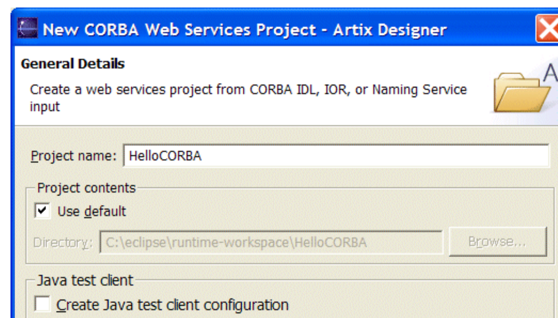


Figure 12: *CORBA Web Services Project Details*

4. Enter a name for your project.
5. If you do not wish to store your project in the default workspace, remove the check from the **Use default** box and enter a new directory into which your project will be stored.
6. If you want to create a code generation profile for a Java test client, select the **Create Java test client configuration** check box.

7. Click **Next** to open the **Inputs and Web Service Options** window, shown in Figure 13.

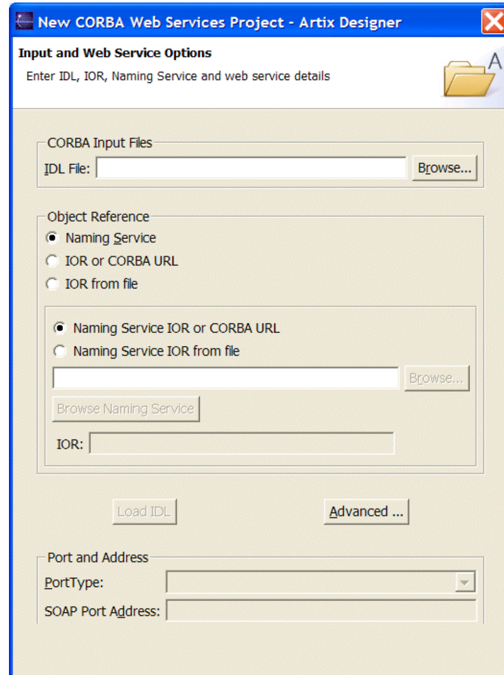


Figure 13: *CORBA Web Services Project Input Details*

8. Enter the location of the IDL file that defines the CORBA service's interface in **IDL File**.

Specifying the CORBA object reference

You specify in the IOR for the CORBA server process in the Object Reference section.

You can do this in one of the following ways:

- Selecting the **Naming Service** radio button, and then Specifying the naming service reference.
- Selecting the **IOR or CORBA URL** radio button and pasting either an IOR string or a corbaloc URL into the **IOR** text box.

- Selecting the **IOR from file** radio button and clicking **Browse** to locate an IOR file or a text file containing an IOR. This will populate the IOR text box.

Specifying the naming service reference

If you selected **Naming Service** in the Object Reference section, you now need to specify the location of the CORBA naming service.

You can do this in one of the following ways:

- Select the **Naming Service IOR** or **CORBA URL** radio button and paste an IOR string or corbaloc URL for the naming service into the text box.
- Populate the text box by selecting the **Naming Service IOR from file** radio button, then clicking **Browse** to select an IOR file or a text file containing an IOR.

Once you have located the naming service, click the **Browse Naming Service** button to display the Naming Service Contexts dialog box, as shown in Figure 14.

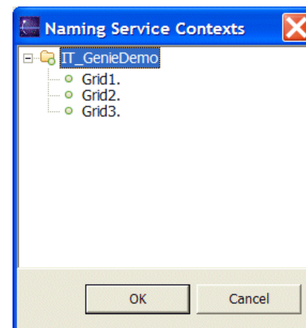


Figure 14: *The Naming Service Contexts Dialog Box*

Select the IOR for the CORBA server process and then click **OK**. This will populate the IOR text box.

Adding the Web service details

Once the **IDL** and **IOR** text boxes have been populated, click the **Load IDL** button to populate the Port and Address section.

In the Port and Address section, select the port type corresponding to the desired CORBA servant from the **PortType** list.

You can also edit the value assigned to the **SOAP Port Address** text box if you wish.

Creating a Project Using a Template

Overview

When creating a basic Web services project, you have the option of using a template. The templates help you create the elements needed for specific types of application. They prompt you for the contracts to import and set-up code generation profiles using default settings for you. Template-generated projects include the starting point code and other elements needed to deploy and test an Artix solution.

Template types

There are a total of seven templates that you can use when generating a basic Web services project. These can be divided up into three broad categories:

- The Artix switch template
- C++ templates
- Java templates

The Artix switch template

The Artix switch template creates a directory structure for a basic web services project and adds generation and configuration details for an Artix switch.

The generated project will contain a folder under `outputs\switches` that contains start and stop scripts for an Artix switch along with a configuration domain under which the switch will run.

C++ templates

There are three types of C++ template available:

- C++ client
- C++ server
- C++ client and server

In each case, the generated project will contain a folder under `outputs\applications` containing the following:

- Start scripts for the compiled application or applications
- A configuration domain under which the applications will run
- The C++ code generated from the specified WSDL file
- The client and/or server executables

Artix will create the code generation profile for a template project using the default values set in the Artix Designer C++ preferences window. These settings determine the namespace under which the C++ code is generated and the default C++ declaration specification in the code.

The default setting for the C++ namespace is `COM_IONA_ARTIX`. The default setting for the declaration specification is `ARTIX_PROJECT`.

For information on changing these settings see "Setting Artix Designer Preferences" on page 13.

Java templates

There are three types of Java template available:

- Java client
- Java server
- Java client and server

In each case, the generated project will contain a folder under `outputs\applications` containing the following:

- Start scripts for the compiled application or applications
- A configuration domain under which the applications will run
- The Java code generated from the specified WSDL file

There will also be an additional folder, `bin`, added to your project. This folder will contain the compiled Java classes for your application or applications.

Artix will create the code generation profile for a template project using the default values set in the Artix Designer Java preferences window. These settings determine the default package name under which the Java code is generated and if the generated code will overwrite any existing code.

The package name for generated code is determined by adding the project name onto the end of the default package name specified in the preference window. The default setting for the Java package name is `com.iona.artix`.

For information on changing these settings see "Setting Artix Designer Preferences" on page 13.

Creating a project from a template

To create an Artix Designer project using a template:

1. Select **File|New|Project** to open the new project wizard.
2. Select **Basic Web Services Project** under the **IONA Artix Designer** folder.

3. Enter a project name.
4. If you do not wish to store your project in the default workspace, remove the check from the **Use default** box and enter a new directory into which your project will be stored.
5. Place check in the **Use a template** box to make the templates available.
6. Select a template from the list of available templates.
7. Enter the name of the WSDL file the defines the services and routes to be connected.
8. If you are creating an Artix switch, proceed to step **11**.
9. From the **Service** drop-down list, select the name of the WSDL service from which to generate code.
10. From the **Port** drop-down list, select the name of the WSDL port on which your server will listen for requests.
11. Select **Finish** to create the project.

Creating Artix Resources

An Artix resource is an artifact that defines a piece of an Artix solution.

In this chapter

This chapter discusses the following topics:

What are Artix Resources?	page 32
Creating Design Resources	page 34
Working with Generation Profiles	page 85

What are Artix Resources?

Overview

An Artix resource is an artifact that defines a piece of an Artix solution. In Artix there are several types of Artix resources:

- Contracts define the services of an Artix solution.
- Schemas define the data types used by the services.
- Access control lists define security roles for services.
- Generation profiles define how Artix creates code and configuration files from contracts and schemas.
- Java objects define the physical implementation of a service.
- C++ objects define the physical implementation of a service.
- Configuration files define the runtime behavior of the deployed Artix-enabled services.

Using the WSDL editor

Artix Designer's WSDL editor provides wizards that guide you through the process of adding elements to both schemas and full contracts. The wizards simplify the process of building complex type definitions and interface definitions by prompting you for only the relevant information. The wizards also prevent you from entering invalid information.

Using the Artix Generator

Artix code generation profiles allow you to specify the types of deployment resources Artix will generate for you. They also allow you to specify what Artix services will be deployed as part of a particular application. For example, you can create a generation profile for a Java client and another one for a C++ service based on the same contract.

Table 1 shows the type of code generation profiles that are available in the Artix Generator.

Table 1: *Artix Code Generation Profiles*

Profile	Description
Container	Provides scripts for starting and stopping the Artix container. It also creates a configuration file for deploying an empty container.

Table 1: *Artix Code Generation Profiles*

Profile	Description
Service Plug-in	Generates code for implementing a service as an Artix plug-in. In addition, it generates configuration and scripts for deploying the plug-in into the Artix container.
Application	Generates code for developing an Artix-enabled application. You can specify if the application is a client or a server. In addition, it generates a configuration file and scripts for deploying the application.
CORBA IDL	Generates IDL from a contract that contains a CORBA binding. In addition, it generates an environment script to source the Artix environment and a configuration file.
Switch	Generates a start script and configuration file for deploying an Artix router inside of the Artix container. The deployed switch will use the specified contract as the source of its routing rules.
Types	Generates the code needed to support the types defined in a contract or schema document. In addition, it generates a environment script for sourcing the Artix environment and a configuration file template for a switch.

Creating Design Resources

Overview

The first step in designing an Artix solution is to create contracts and schemas to define your services. Artix designer provides wizards to guide through the process of creating a contract shell and adding the needed definitions to it.

Artix also provides a number of tools to create contracts from existing service resources including existing WSDL, IDL files, Java classes, and COBOL copybooks.

In this section

This section discusses the following topics:

Creating a New Contract	page 35
Importing a Contract from a URL	page 38
Creating a Contract from CORBA IDL	page 40
Creating a Contract from a Java Class	page 50
Creating a Contract from a COBOL Copybook	page 62
Creating a Contract from a Data Set	page 69
Creating a Contract from an XML Schema Document	page 78
Creating an XML Schema	page 80
Importing an XML Schema from a URL	page 82
Creating Access Control Lists	page 83

Creating a New Contract

Overview

When creating a new contract using the **WSDL File** wizard you have two options:

- Creating an empty contract and the filling in the details.
- Linking to an existing WSDL document on a locally accessible file system.

Creating an empty contract

When you create an empty contract, the new contract is placed under the project's file system.

To add an empty contract to your project:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15.

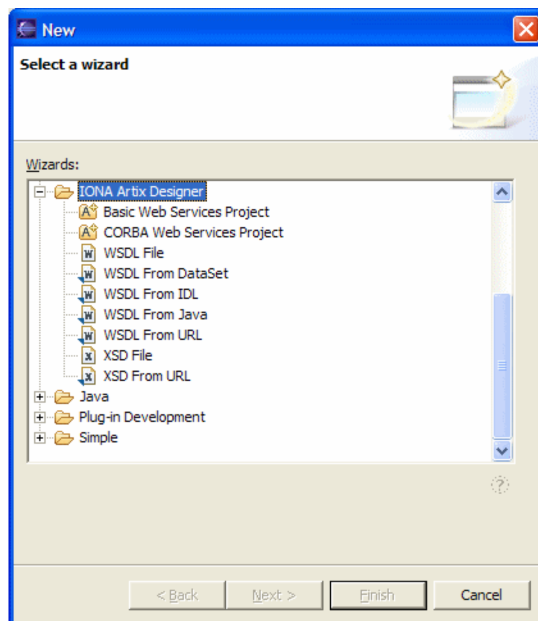


Figure 15: *The New Window*

2. Select **WSDL File** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **WSDL File** window, shown in Figure 16.

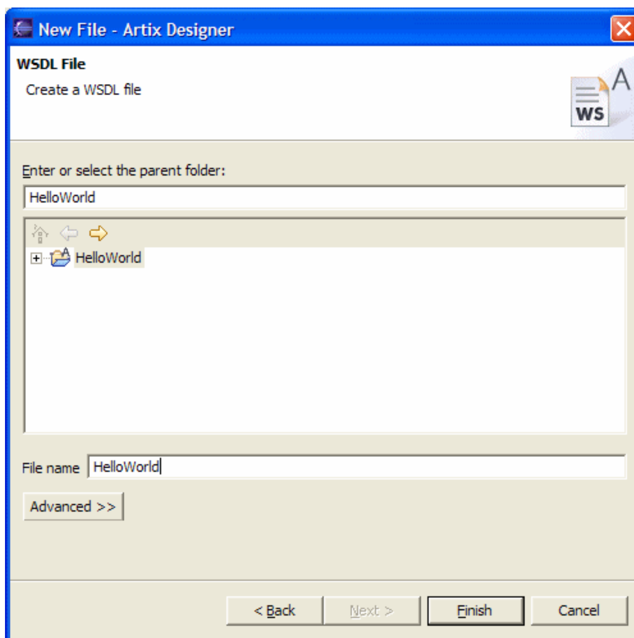


Figure 16: *Entering WSDL Details*

4. Select the folder where you want to store the contract.
5. Type the name of the file in the **File name** field.
6. Click **Finish**.

Linking to an existing WSDL

When you link to an existing WSDL document, you create an entry in your project for the WSDL document. The entry in your project is not a copy of the original file. It is a hard link to the original document and any changes made to the document are reflected in the original WSDL document.

To link to an existing WSDL document:

1. From the **File** menu, select **New|Other**.
2. Select **WSDL File** from under the **IONA Artix Designer** folder.

3. Click **Next**.
4. Select the folder where you want to store the contract.
5. Type the name of the file in the **File name** field.
6. Click **Advanced** to expand the **WSDL File** window as shown in Figure 17.

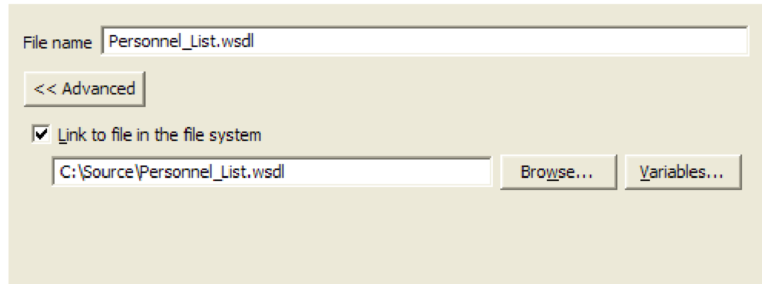


Figure 17: *Entering WSDL Link Details*

7. Select the **Link to file in the file system** check box.
8. Click the **Browse** button to select the WSDL file that you want to link to.
9. Click **Finish**.

Importing a Contract from a URL

Overview

If you want to create a client or service from an existing WSDL document that is posted at a URL, you can import a read-only version of the WSDL document into an Artix Designer project.

Once imported, the contract can be used as an import for other contracts, a target for an Artix switch, or as a basis for generating code.

Procedure

To import a contract from a URL:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15.
2. Select **WSDL From URL** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **WSDL File** window, shown in Figure 18.

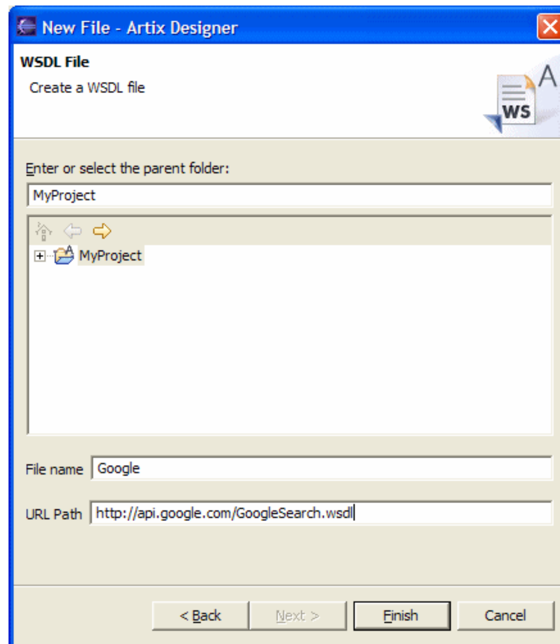


Figure 18: *Entering WSDL URL Details*

4. Select the folder where you want to store the contract.
5. Type the name of the contract in the **File name** field.
6. Type the URL in the **URL Path** field.
7. Click **Finish**.

Creating a Contract from CORBA IDL

Overview

If you are starting from a CORBA server or client, Artix can build the logical portion of the Artix contract from IDL. Contracts generated from IDL have CORBA-specific entries and namespaces added. The IDL to WSDL compiler also generates the binding information required to format the operations specified in the IDL. However, since port information is specific to the deployment environment, the port information is left blank.

Artix provides two methods of generating Artix contracts from an IDL file:

- Using Artix Designer
- Using the idltowsdl tool

CORBA WSDL namespaces

Contracts generated from IDL include two additional name spaces:

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

Unsupported type handling

Be aware that the IDL to WSDL compiler ignores any definitions that use unsupported CORBA types. The IDL to WSDL compiler also ignores any definition that uses a previously ignored definition. For example, assume you have the following IDL definitions in `file.idl`:

```
interface A
{
  struct S
  {
    wchar member;
  };

  S get_op();
};
```

The IDL to WSDL compiler does not generate any corresponding contract information for the structure `S` because it contains a member that uses an unsupported type. Similarly, the IDL to WSDL compiler does not generate any contract information for the operation `get_op()` because it references structure `S`.

Using Artix Designer

To use an IDL file as the basis for a contract:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15.
2. Select **WSDL From IDL** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **WSDL File** window, shown in Figure 19.

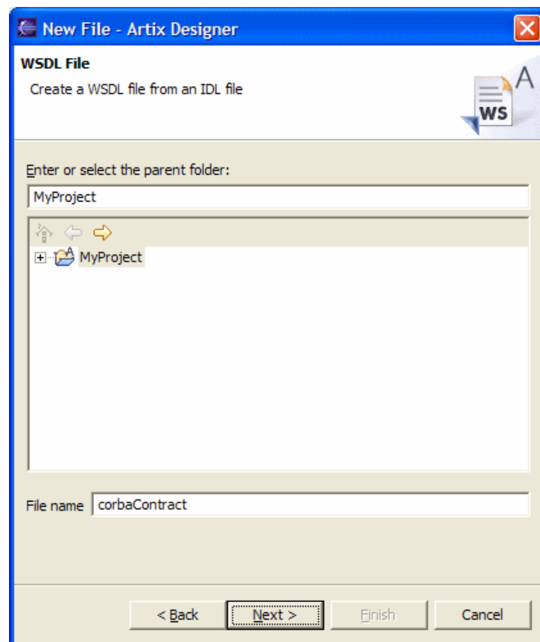


Figure 19: *Entering Contract Name*

4. Select the folder where you want to store the WSDL file.
5. Type the name of the file in the **File name** field.

6. Click **Next** to open the **Select IDL File** window, shown in Figure 20.

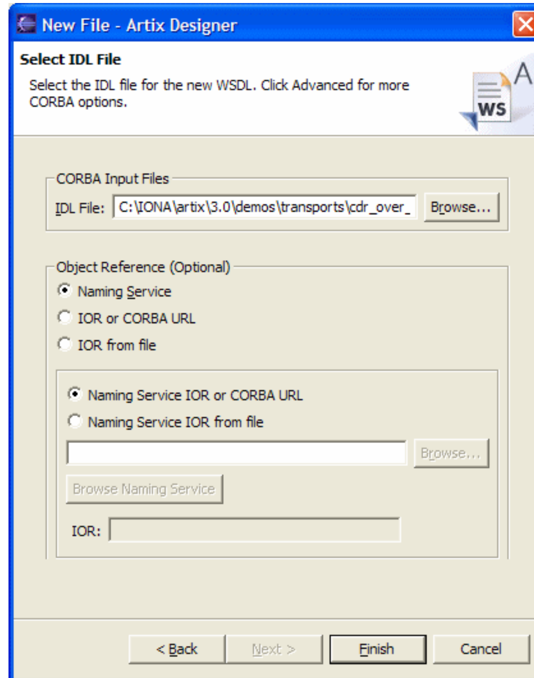


Figure 20: *Specifying CORBA IDL Details*

7. Enter the location of the IDL file that defines the CORBA service's interface in **IDL File**.

Specifying the CORBA object reference

You specify in the IOR for the CORBA server process in the Object Reference section.

You can do this in one of the following ways:

- Selecting the **Naming Service** radio button, and then Specifying the naming service reference.
- Selecting the **IOR or CORBA URL** radio button and pasting either an IOR string or a corbaloc URL into the **IOR** text box.

- Selecting the **IOR from file** radio button and clicking **Browse** to locate an IOR file or a text file containing an IOR. This will populate the IOR text box.

Specifying the naming service reference

If you selected **Naming Service** in the Object Reference section, you now need to specify the location of the CORBA naming service.

You can do this in one of the following ways:

- Select the **Naming Service IOR** or **CORBA URL** radio button and paste an IOR string or corbaloc URL for the naming service into the text box.
- Populate the text box by selecting the **Naming Service IOR from file** radio button, then clicking **Browse** to select an IOR file or a text file containing an IOR.

Once you have located the naming service, click the **Browse Naming Service** button to display the Naming Service Contexts dialog box, as shown in Figure 14.

Select the IOR for the CORBA server process and then click **OK**. This will populate the IOR text box.

Using the idltowsdl tool

IONA's IDL to WSDL compiler supports several command line flags that specify how to create a WSDL file from an IDL file. The default behavior of the tool is to create WSDL file that uses wrapped doc/literal style messages. Wrapped doc/literal style messages have a single part, defined using an element, that wraps all of the elements in the message. See Example 2 on page 47 for a sample.

The IDL to WSDL compiler is run using the following command:

```
idltowsdl  [-usetypes] [-unwrap] [-a address] [-f file] [-o dir] [-s
    type] [-r file] [-L file] [-P file]  [-w namespace]
    [-x namespace] [-t namespace] [-T file] [-n file] [-b] [-I idlDir]
    [-qualified] [-inline] [-3] [-fasttrack] [-interface name]
    [-soapaddr port] [-L file]  [-q] [-h] [-V]  idlfile
```

The command has the following options:

`-usetypes`	Generate rpc style messages. rpc style messages have parts defined using XML Schema types instead of XML elements.

`-unwrap`	Generate unwrapped doc/literal messages. Unwrapped messages have parts that represent individual elements. Unlike wrapped messages, unwrapped messages can have multiple parts and are not allowed by the WS-I.
`-a` *`address`*	Specifies an absolute address through which the object reference may be accessed. The *`address`* may be a relative or absolute path to a file, or a corbaname URL
`-f` *`file`*	Specifies a file containing a string representation of an object reference. The object reference is placed in the `corba:address` element in the `port` definition of the generated service. The *`file`* must exist when you run the IDL compiler.
`-o` *`dir`*	Specifies the directory into which the WSDL file is written.
`-s` *`type`*	Specifies the XML Schema type used to map the IDL `sequence<octet>` type. Valid values are `base64Binary` and `hexBinary`. The default is `base64Binary`.
`-r` *`file`*	Specify the pathname of the schema file imported to define the `Reference` type. If the `-r` option is not given, the idl compiler gets the schema file pathname from `etc/idl.cfg`.
`-L` *`file`*	Specifies that the logical portion of the generated WSDL specification into is written to *`file`*. *`file`* is then imported into the default generated file.
`-P` *`file`*	Specifies that the physical portion of the generated WSDL specification into is written to *`file`*. *`file`* is then imported into the default generated file.
`-w` *`namespace`*	Specifies the namespace to use for the WSDL `targetNamespace`. The default is `http://schemas.iona.com/idl/`*`idl_name`*.
`-x` *`namespace`*	Specifies the namespace to use for the Schema `targetNamespace`. The default is `http://schemas.iona.com/idltypes/`*`idl_name`*.
`-t` *`namespace`*	Specifies the namespace to use for the CORBA TypeMapping `targetNamespace`. The default is `http://schemas.iona.com/typemap/corba/`*`idl_name`*.

`-T` *`file`*	Specifies that the schema types are to be generated into a separate file. The schema file is included in the generated contract using an import statement. This option cannot be used with the `-n` option.
`-n` *`file`*	Specifies that a schema file, *`file`*, is to be included in the generated contract by an import statement. This option cannot be used with the `-T` option.
`-b`	Specifies that bounded strings are to be treated as unbounded. This eliminates the generation of the special types for the bounded string.
`-I` *`idlDir`*	Specify a directory to be included in the search path for the IDL preprocessor. You can use this flag multiple times.
`-qualified`	Generates fully qualified WSDL.
`-inline`	Generates a contract that includes all imported documents in-line. This overrides all options that specify that a section of the contract is to be imported.
`-3`	Use relaxed IDL grammar checking semantics to allow IDL used by Orbix 3 to be parsed.
`-fasttrack`	Use the fasttrack wizard. You must also use the `-interface` and `-soapaddr` flags with this option. This option also adds a SOAP port and a route between the generated CORBA port and the generated SOAP port.
`-interface` *`name`*	Specifies the IDL interface for which WSDL will be generated by the fastrack wizard.
`-soapaddr` *`port`*	Specifies the SOAP address to use in the generated `port` element when using the fasttrack wizard.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

To combine multiple flags in the same command, use a colon-delimited list. The colon is only interpreted as a delimiter if it is followed by a dash. Consequently, the colons in a `corbaname` URL are interpreted as part of the URL syntax and not as delimiters.

Note: The command line flag entries are case sensitive even on Windows. Capitalization in your generated WSDL file must match the capitalization used in the prewritten code.

Example

Imagine you needed to generate an Artix contract for a CORBA server that exposes the interface shown in Example 1.

Example 1: *personalInfoService Interface*

```
interface personalInfoService
{
  enum hairColorType {red, brunette, blonde};
  struct personalInfo
  {
    string name;
    long age;
    hairColorType hairColor;
  };
  exception idNotFound
  {
    short id;
  };
  personalInfo lookup(in long empId)
  raises (idNotFound);
};
```

To generate the contract, you run it through the IDL compiler using either Artix Designer or the command line. The resulting contract is similar to that shown in Example 2.

Example 2: *personalInfoService Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="personalInfo.idl"
 targetNamespace="http://schemas.iona.com/idl/personalInfo.idl"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.iona.com/idl/personalInfo.idl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://schemas.iona.com/idltypes/personalInfo.idl"
 xmlns:corba="http://schemas.iona.com/bindings/corba"
 xmlns:corbatm="http://schemas.iona.com/typemap/corba/personalInfo.idl"
 xmlns:references="http://schemas.iona.com/references">
  <types>
    <schema targetNamespace="http://schemas.iona.com/idltypes/personalInfo.idl"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:simpleType name="personalInfoService.hairColorType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="red"/>
          <xsd:enumeration value="brunette"/>
          <xsd:enumeration value="blonde"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:complexType name="personalInfoService.personalInfo">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="age" type="xsd:int"/>
          <xsd:element name="hairColor" type="xsd1:personalInfoService.hairColorType"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="personalInfoService.idNotFound">
        <xsd:sequence>
          <xsd:element name="id" type="xsd:short"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="personalInfoService.lookup">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="empId" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

Example 2: *personalInfoService Contract*

```
    <xsd:element name="personalInfoService.lookupResult">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="return" type="xsd1:personalInfoService.personalInfo"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="personalInfoService.idNotFound"
                 type="xsd1:personalInfoService.idNotFound"/>
  </schema>
</types>
<message name="personalInfoService.lookup">
  <part name="parameters" element="xsd1:personalInfoService.lookup"/>
</message>
<message name="personalInfoService.lookupResponse">
  <part name="parameters" element="xsd1:personalInfoService.lookupResult"/>
</message>
<message name="personalInfoService.idNotFound">
  <part name="exception" element="xsd1:personalInfoService.idNotFound"/>
</message>
<portType name="personalInfoService">
  <operation name="lookup">
    <input message="tns:personalInfoService.lookup" name="lookup"/>
    <output message="tns:personalInfoService.lookupResponse" name="lookupResponse"/>
    <fault message="tns:personalInfoService.idNotFound" name="personalInfoService.idNotFound"/>
  </operation>
</portType>
<binding name="personalInfoServiceBinding" type="tns:personalInfoService">
  <corba:binding repositoryID="IDL:personalInfoService:1.0"/>
  <operation name="lookup">
    <corba:operation name="lookup">
      <corba:param name="empId" mode="in" idltype="corba:long"/>
      <corba:return name="return" idltype="corbatm:personalInfoService.personalInfo"/>
      <corba:raises exception="corbatm:personalInfoService.idNotFound"/>
    </corba:operation>
    <input/>
    <output/>
    <fault name="personalInfoService.idNotFound"/>
  </operation>
</binding>
<service name="personalInfoServiceService">
  <port name="personalInfoServicePort" binding="tns:personalInfoServiceBinding">
    <corba:address location="..."/>
  </port>
</service>
```

Example 2: *personalInfoService Contract*

```
  <corba:typeMapping targetNamespace="http://schemas.iona.com/typemap/corba/personalInfo.idl">
    <corba:enum name="personalInfoService.hairColorType"
                type="xsd1:personalInfoService.hairColorType"
                repositoryID="IDL:personalInfoService/hairColorType:1.0">
      <corba:enumerator value="red"/>
      <corba:enumerator value="brunette"/>
      <corba:enumerator value="blonde"/>
    </corba:enum>
    <corba:struct name="personalInfoService.personalInfo"
                  type="xsd1:personalInfoService.personalInfo"
                  repositoryID="IDL:personalInfoService/personalInfo:1.0">
      <corba:member name="name" idltype="corba:string"/>
      <corba:member name="age" idltype="corba:long"/>
      <corba:member name="hairColor" idltype="corbatm:personalInfoService.hairColorType"/>
    </corba:struct>
    <corba:exception name="personalInfoService.idNotFound"
                     type="xsd1:personalInfoService.idNotFound"
                     repositoryID="IDL:personalInfoService/idNotFound:1.0">
      <corba:member name="id" idltype="corba:short"/>
    </corba:exception>
  </corba:typeMapping>
</definitions>
```

Creating a Contract from a Java Class

Overview

To facilitate the integration of Java applications with Artix, Artix provides the following tools for generating the logical portion of an Artix contract from existing Java classes:

- Using Artix Designer
- Using the javatowsdl tool

These tools use the mapping rules described in Sun's JAX-RPC 1.1 specification.

Using Artix Designer

This functionality employs Eclipse's Java Development Tools (JDT). In order to use it, your Artix Designer project will need to be Java-enabled, and the class present on the build path.

Creating WSDL from Java is a three-step process:

1. Java-enabling your project
2. Adding the class to the build path
3. Running the WSDL from Java wizard

Java-enabling your project

Your Artix Designer project is Java-enabled as soon as you use Artix Generator to generate some Java code.

If you have not already run a Java code configuration from your project, you should do so now. The quickest way to do this is to create an empty WSDL file and run the Types code generation configuration, since this does not require the WSDL to contain any elements.

Adding the class to the build path

To add the Java class to your project's build path:

1. Package the class into a JAR file and save it somewhere on your hard drive.
2. Right-click your Artix project and select **Properties** from the pop-up menu.
3. In the Java Build Path page, click the **Libraries** tab.
4. Click the **Add External JARs** button.

5. In the JAR Selection dialog box, browse to the location of the JAR file and select it.
6. Click **OK**.

Running the WSDL from Java wizard

To create a contract from a Java class using Artix Designer:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15 on page 35.
2. Select **WSDL From Java** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **WSDL File** window, shown in Figure 21.

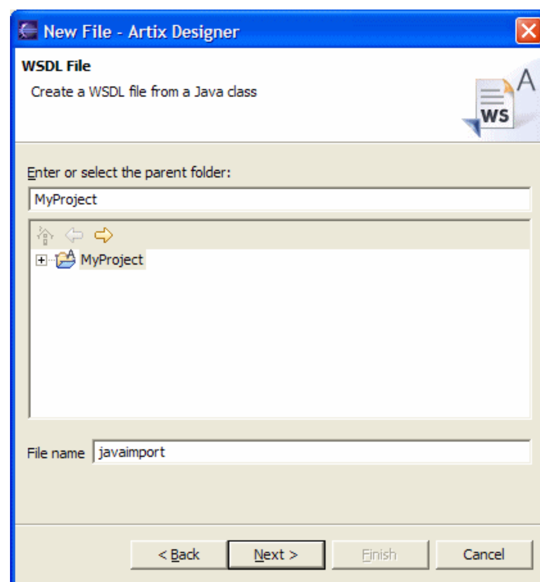


Figure 21: *Entering Contract Name for Java Import*

4. Select the folder where you want to store the WSDL file.
5. Type the name of the file in the **File name** field.

6. Click **Next** to open the **Java Class File** window, shown in Figure 22.

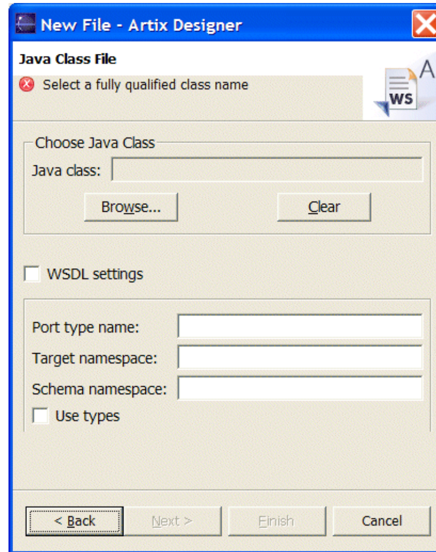


Figure 22: *Java Class Details*

7. Click **Browse** to open the Find Class dialog box shown in Figure 23.

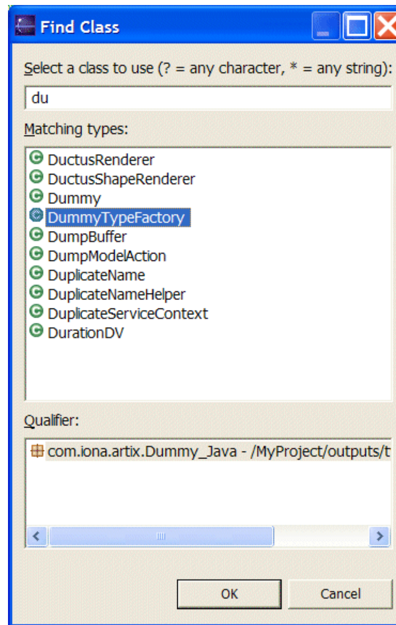


Figure 23: *The Find Class Dialog Box*

8. Enter a search string to locate the class you wish to import in the **Select a class to use** field.
9. Select the desired class from the list of types in the **Matching types** list.
10. Click **OK** to return to the **Java Class File** window.
11. Select the **WSDL settings** check box if you do not want Artix to use default values in the generated contract.
    i. Enter the name you want to use for the generated `portType` element in the **Port type name:** field.
    ii. Enter the namespace you wish to use as the generated contract's target namespace in the **Target namespace:** field.

iii. Enter the namspace you wish to use as the target namespace for the generated contract's `schema` element in the **Schema namespace:** field.

iv. Select the **Use types** check box if you want Artix to generate the contract using types in the message parts.

12. Click **Finish**.

Using the javatowsdl tool

Artix supplies a command line tool, `javatowsdl`, that generates the logical portion of an Artix contract for existing Java class files. To generate the logical portion of an Artix contract using the `javatowsdl` tool use the following command:

```
javatowsdl  [-t namespace] [-x namespace] [-i porttype]
            [-o file] [-useTypes] [-v] [-?] [-L file]
            [-q] [-h] [-V]  ClassName
```

The command has the following options:

`-t` *namespace*	Specifies the target namespace of the generated WSDL document. By default, the java package name will be used as the target namespace. If no package name is specified, the generated target namespace will be `http:\\www.iona.com\`*ClassName*.
`-x` *namespace*	Specifies the target namespace of the XML Schema information generated to represent the data types inside the WSDL document. By default, the generated target namespace of the XML Schema will be `http:\\www.iona.com\`*ClassName*`\xsd`.
`-i` *porttype*	Specifies the name of the generated `portType` in the WSDL document. By default, the name of the class from which the WSDL is generated is used.
`-o` *file*	Specifies output file into which the WSDL is written.
`-useTypes`	Specifies that the generated WSDL will use types in the WSDL message parts. By default, messages are generated using wrapped `doc/literal` style. A wrapper element with a sequence will be created to hold method parameters.
`-v`	Prints out the version of the tool.

`-?`	Prints out a help message explaining the command line flags.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

The generated WSDL will not contain any physical details concerning the payload formats or network transports that will be used when exposing the service. You will need to add this information manually.

Note: When generating contracts, `javatowsdl` will add newly generated WSDL to an existing contract if a contract of the same name exists. It will not generate a new file or warn you that a previous contract exists.

Supported types

Table 2 shows the Java types Artix can map to an Artix contract.

Table 2: *Java to WSDL Mappings*

Java	Artix Contract
`boolean`	`xsd:boolean`
`byte`	`xsd:byte`
`short`	`xsd:short`
`int`	`xsd:int`
`long`	`xsd:long`
`float`	`xsd:float`
`double`	`xsd:double`
`byte[]`	`xsd:base64binary`
`java.lang.String`	`xsd:string`
`java.math.BigInteger`	`xsd:integer`

Table 2: *Java to WSDL Mappings (Continued)*

Java	Artix Contract
`java.math.BigDecimal`	`xsd:decimal`
`java.util.Calendar`	`xsd:dateTime`
`java.util.Date`	`xsd:dateTime`
`java.xml.namespace.QName`	`xsd:QName`
`java.net.URI`	`xsd:anyURI`

In the case of helper classes for a Java primitive, such as `java.lang.Integer`, the instance is mapped to an element with the nillable attribute set to true and the type set to the corresponding Java primitive type. Example 3 shows the mapping for a `java.lang.Float`.

Example 3: *Mapping of java.lang.Float to XML Schema*

```
<element name="floatie" nillable="true" type="xsd:float"/>
```

Exceptions

Artix will map user-defined exceptions to the logical Artix contract according to the rules laid out in the JAX-RPC specification. The exception will be mapped to a `fault` element within the operation representing the corresponding Java method. The generated `fault` element will reference a generated `message` element describing the Java exception class. The name attribute of the `message` element will be taken from the name of the Java exception class.

Because SOAP only supports `fault` messages with a single `part`, the generated `message` element is mapped to have only one `part`. When the Java exception only has one field, it is used as the `part` and its `name` and `type` attributes are mapped from the exception’s field. When the Java exception contains more than one field, Artix generates a `complexType` to describe the exception’s data. The generated `complexType` will have one

element for each field of the exception. The `name` and `type` attributes of the generated element will be taken from the corresponding field in the exception.

Note: Standard Java exceptions are not mapped into the generated Artix contract.

Example

For example, if you had a Java interface similar to that shown in Example 4, you could generate an Artix contract on it by compiling the interface into a `.class` file and running the command `javatowsdl Base`.

Example 4: *Base Java Class*

```
//Java
public interface Base
{
  public byte[] echoBase64(byte[] inputBase64);
  public boolean echoBoolean(boolean inputBoolean);
  public float echoFloat(float inputFloat);
  public float[] echoFloatArray(float[] inputFloatArray);
  public int echoInteger(int inputInteger);
  public int[] echoIntegerArray(int[] inputIntegerArray);
}
```

The resulting Artix contract will be similar to Example 5.

Example 5: *Base Artix Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="Base" targetNamespace="http://www.iona.com/Base"
    xmlns:ns1="http://www.iona.com/Base" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://www.iona.com/Base/xsd">
    <wsdl:types>
      <schema targetNamespace="http://www.iona.com/Base/xsd"
              xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="echoBoolean">
          <complexType>
            <sequence>
              <element name="booleanParam0" type="xsd:boolean"/>
            </sequence>
          </complexType>
        </element>
```

Example 5: *Base Artix Contract (Continued)*

```
        <element name="echoBooleanResponse">
          <complexType>
            <sequence>
              <element name="return" type="xsd:boolean"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoBase64">
          <complexType>
            <sequence>
              <element maxOccurs="unbounded" minOccurs="0" name="_bParam0" type="xsd:byte"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoBase64Response">
          <complexType>
            <sequence>
              <element maxOccurs="unbounded" minOccurs="0" name="return" type="xsd:byte"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoHexBinary">
          <complexType>
            <sequence>
              <element maxOccurs="unbounded" minOccurs="0" name="_bParam0" type="xsd:byte"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoHexBinaryResponse">
          <complexType>
            <sequence>
              <element maxOccurs="unbounded" minOccurs="0" name="return" type="xsd:byte"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoFloat">
          <complexType>
            <sequence>
              <element name="floatParam0" type="xsd:float"/>
            </sequence>
          </complexType>
        </element>
```

Example 5: *Base Artix Contract (Continued)*

```
        <element name="echoFloatResponse">
          <complexType>
            <sequence>
              <element name="return" type="xsd:float"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoFloatArray">
          <complexType>
            <sequence>
               <element maxOccurs="unbounded" minOccurs="0" name="_fParam0" type="xsd:float"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoFloatArrayResponse">
          <complexType>
            <sequence>
              <element maxOccurs="unbounded" minOccurs="0" name="return" type="xsd:float"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoInteger">
          <complexType>
            <sequence>
              <element name="intParam0" type="xsd:int"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoIntegerResponse">
          <complexType>
            <sequence>
              <element name="return" type="xsd:int"/>
            </sequence>
          </complexType>
        </element>
        <element name="echoIntegerArray">
          <complexType>
            <sequence>
              <element maxOccurs="unbounded" minOccurs="0" name="_iParam0" type="xsd:int"/>
            </sequence>
          </complexType>
        </element>
```

Example 5: *Base Artix Contract (Continued)*

```
      <element name="echoIntegerArrayResponse">
        <complexType>
          <sequence>
            <element maxOccurs="unbounded" minOccurs="0" name="return" type="xsd:int"/>
          </sequence>
        </complexType>
      </element>
    </wsdl:types>
    <wsdl:message name="echoBoolean">
      <wsdl:part element="xsd1:echoBoolean" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoBooleanResponse">
      <wsdl:part element="xsd1:echoBooleanResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoBase64">
      <wsdl:part element="xsd1:echoBase64" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoBase64Response">
      <wsdl:part element="xsd1:echoBase64Response" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoHexBinary">
      <wsdl:part element="xsd1:echoHexBinary" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoHexBinaryResponse">
      <wsdl:part element="xsd1:echoHexBinaryResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloat">
      <wsdl:part element="xsd1:echoFloat" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloatResponse">
      <wsdl:part element="xsd1:echoFloatResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloatArray">
      <wsdl:part element="xsd1:echoFloatArray" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloatArrayResponse">
      <wsdl:part element="xsd1:echoFloatArrayResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoInteger">
      <wsdl:part element="xsd1:echoInteger" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoIntegerResponse">
      <wsdl:part element="xsd1:echoIntegerResponse" name="parameters"/>
    </wsdl:message>
```

Example 5: *Base Artix Contract (Continued)*

```
    <wsdl:message name="echoIntegerArray">
      <wsdl:part element="xsd1:echoIntegerArray" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoIntegerArrayResponse">
      <wsdl:part element="xsd1:echoIntegerArrayResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:portType name="Base">
      <wsdl:operation name="echoBoolean">
        <wsdl:input message="ns1:echoBoolean" name="echoBoolean"/>
        <wsdl:output message="ns1:echoBooleanResponse" name="echoBoolean"/>
      </wsdl:operation>
      <wsdl:operation name="echoBase64">
        <wsdl:input message="ns1:echoBase64" name="echoBase64"/>
          <wsdl:output message="ns1:echoBase64Response" name="echoBase64"/>
      </wsdl:operation>
      <wsdl:operation name="echoHexBinary">
        <wsdl:input message="ns1:echoHexBinary" name="echoHexBinary"/>
        <wsdl:output message="ns1:echoHexBinaryResponse" name="echoHexBinary"/>
      </wsdl:operation>
      <wsdl:operation name="echoFloat">
        <wsdl:input message="ns1:echoFloat" name="echoFloat"/>
        <wsdl:output message="ns1:echoFloatResponse" name="echoFloat"/>
      </wsdl:operation>
      <wsdl:operation name="echoFloatArray">
        <wsdl:input message="ns1:echoFloatArray" name="echoFloatArray"/>
        <wsdl:output message="ns1:echoFloatArrayResponse" name="echoFloatArray"/>
      </wsdl:operation>
      <wsdl:operation name="echoInteger">
        <wsdl:input message="ns1:echoInteger" name="echoInteger"/>
        <wsdl:output message="ns1:echoIntegerResponse" name="echoInteger"/>
      </wsdl:operation>
      <wsdl:operation name="echoIntegerArray">
        <wsdl:input message="ns1:echoIntegerArray" name="echoIntegerArray"/>
        <wsdl:output message="ns1:echoIntegerArrayResponse" name="echoIntegerArray"/>
      </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

Creating a Contract from a COBOL Copybook

Overview

To facilitate the mapping of COBOL operations to Artix contracts, Artix provides the following tools for generating Artix contracts from COBOL copybooks:

- Using Artix Designer
- Using the coboltowsdl tool

The generated contracts contain a fixed binding to define the COBOL interface for Artix applications.

Using Artix Designer

To add a contract containing a fixed binding from a COBOL copybook:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15.
2. Select **WSDL From DataSet** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **WSDL File** window.
4. Select the folder where you want to store the WSDL file.
5. Type the name of the file in the **File name** field.
6. Click **Next** to open the **Select Data Format** window, shown in Figure 24.

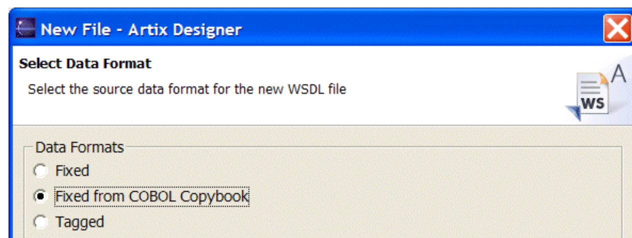


Figure 24: *Select Data Format*

7. Select **Fixed from COBOL Copybook**.

8. Click **Next** to open the **Set Defaults** window, shown in Figure 25.

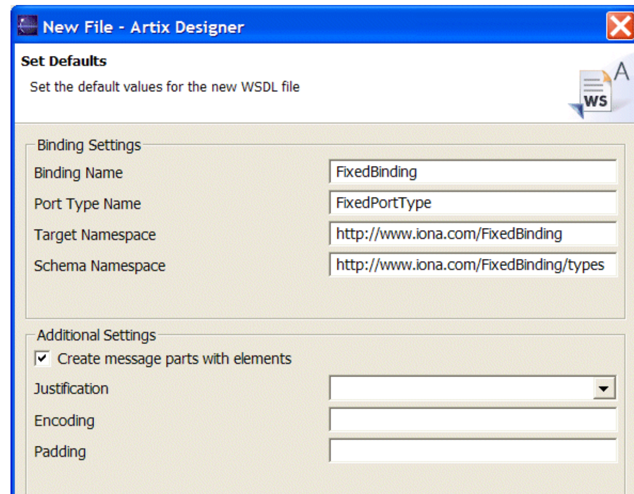


Figure 25: *The Set Defaults Window*

9. Enter the name for the generated `binding` element in the **Binding Name** field.
10. Enter the name for the generated `portType` element in the **Port Type Name** field.
11. Enter the namesapce for the generated contract's target namespace in the **Target Namespace** field.
12. Enter the namespace you wish to use as the target namespace for the generated contract's `schema` element in the **Schema Namespace** field.
13. Select the **Create message parts with elements** check box if you want to generate a contract where the message parts are defined using `element` elements.
14. Select the justification value to use in the generated fixed binding from the **Justification** drop-down list.
15. Enter the character encoding you wish to use for the generated fixed binding in the **Encoding** field.
16. Enter the character to use for padding on the wire in the **Padding** field.

17. Click **Next** to open the **Input Data** window, shown in Figure 26.

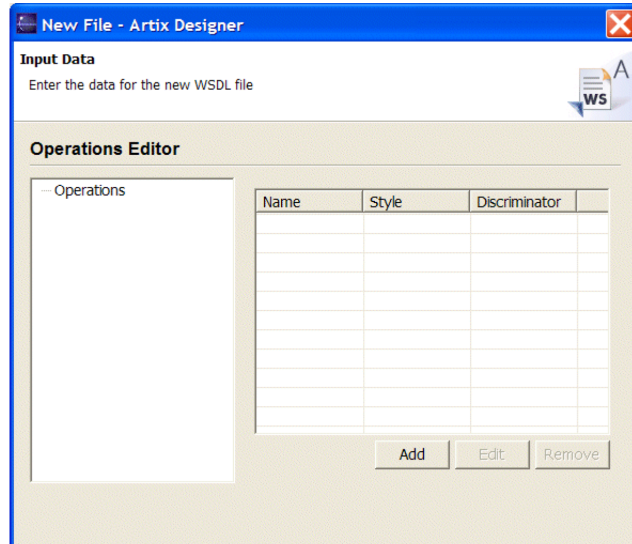


Figure 26: *The Operations Editor*

18. Click **Add** to add a new operation to the table.
19. Select the new operation from the table.

20. Click **Edit** to bring up the operation editing table shown in Figure 27.

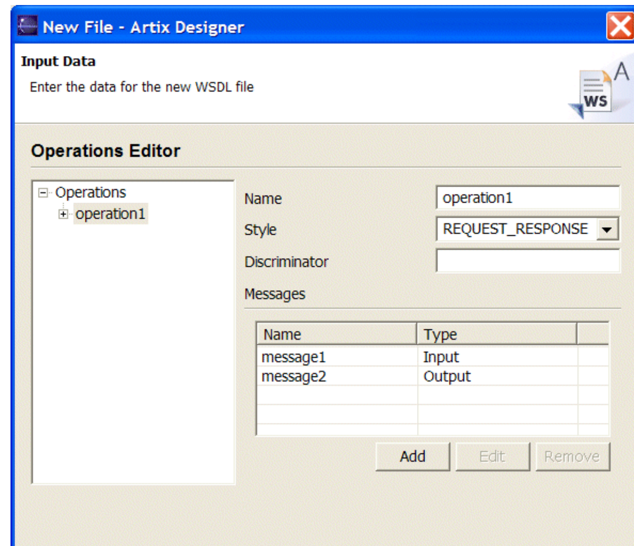


Figure 27: *Editing an Operation*

21. Enter a name for the operation in the **Name** field.
22. Select what type of operation you want to define from the **Style** drop-down list.
    - **REQUEST_RESPONSE** specifies that the operation will use two messages. One for the input message. The other message is for the output message.
    - **ONE_WAY** specifies that the operation will only have one message. The message can be either an input message or an output message.
23. Enter a discriminator string for the operation in the **Discriminator** field. See “fixed:operation” on page 171.

24. Select one of the messages from tree on the left to bring up the message editing table shown in Figure 28.

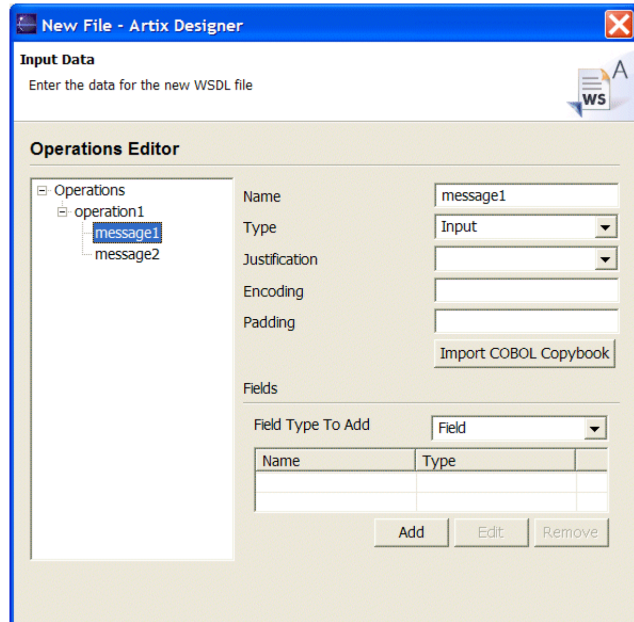


Figure 28: *Editing a Message*

25. Enter a name for the message in the **Name** field.
26. Select if this message is an input message or an output message from the **Type** drop-down list.
27. Enter the desired attributes for the message. See “fixed:body” on page 171.
28. Click **Import COBOL Copybook** to bring up a file browser.
29. Select the copybook that defines the message from the file browser.
30. Repeat steps **24** through **29** for each message in the operation.
31. Repeat steps **18** through **30** for each operation in the service you are defining.
32. Click **Finish**.

Using the coboltowsdl tool

You can also use the Artix command line tools to generate an Artix contract from COBOL copybook data. To do so use the following command:

```
coboltowsdl -b binding -op operation -im [inmessage:]incopybook
            [-om [outmessage:]outcopybook]
            [-fm [faultmessage:]faultbook]
            [-i portType][-t target]
            [-x schema_name][-useTypes][-o file][-L file]
            [-q][-h][-V]
```

The command has the following options:

`-b` *binding*	Specifies the name for the generated binding.
`-op` *operation*	Specifies the name for the generated operation.
`-im [`*inmessage*`:]`*incopybook*	Specifies the name of the input message and the copybook file from which the data defining the message is taken. The input message name, *inmessage*, is optional. However, if the copybook has more than one `01` levels, you will be asked to choose the one you want to use as the input message.
`-om [`*outmessage*`:]`*outcopybook*	Specifies the name of the output message and the copybook file from which the data defining the message is taken. The output message name, *outmessage*, is optional. However, if the copybook has more than one `01` levels, you will be asked to choose the one you want to use as the output message.
`-fm [`*faultmessage*`:]`*faultbook*	Specifies the name of a fault message and the copybook file from which the data defining the message is taken. The fault message name, *faultmessage*, is optional. However, if the copybook has more than one `01` levels, you will be asked to choose the one you want to use as the fault message. You can specify more than one fault message.
`-i` *portType*	Specifies the name of the port type in the generated WSDL. Defaults to *binding*`PortType`.[a]

`-t` *`target`*	Specifies the target namespace for the generated WSDL. Defaults to `http://www.iona.com/`*`binding`*.
`-x` *`schema_name`*	Specifies the namespace for the schema in the generated WSDL. Defaults to `http://www.iona.com/`*`binding`*`/types`.
`-useTypes`	Specifies that the generated WSDL will use `type` elements. Default is to generate `element` elements for schema types.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`binding`*`.wsdl`.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

a. If *`binding`* ends in `Binding` or `binding`, it is stripped off before being used in any of the default names.

Once the new contract is generated, you will still need to add the port information before you can use the contract to develop an Artix solution.

Creating a Contract from a Data Set

Overview

Another way you can create new contracts is by basing them on a data set. Examples of this include:

- Entering a fixed data set
- Entering a tagged data set

When you create a contract in this way, you're actually also creating the associated binding at the same time. When creating contracts using the other methods described in this chapter, the binding definition is a separate step.

Procedure

To create a contract from a data set that is not stored as a COBOL copybook:

1. From the **File** menu, select **New|Other** to open the **New**.
2. Select **WSDL From DataSet** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **WSDL File** window.
4. Select the folder where you want to store the WSDL file.
5. Type the name of the file in the **File name** field.
6. Click **Next** to open the **Select Data Format** window, as shown in Figure 24 on page 62.
7. Select the type of binding you want added to the contract you are creating.
    - **Fixed** will create a contract with a fixed binding. See "Entering a fixed data set" on page 69.
    - **Tagged** will create a contract with a tagged binding. See "Entering a tagged data set" on page 73.

Entering a fixed data set

If you select **Fixed** do the following:

1. Click **Next** to bring up the **Set Defaults** window, shown in Figure 25 on page 63.
2. Enter a name for the generated fixed binding in the **Binding Name** field.

3. Enter a name for the generated `portType` element in the **Port Type Name** field.
4. Enter a target namespace for the generated contract in the **Target Namespace** field.
5. Enter a target namespace for the generated `types` element in the **Schema Namespace** field.
6. If you want the generated message parts to reference elements instead of types, check the **Create message parts with elements** box.
7. Select how you want the data in your binding justified from the **Justification** drop-down list.
8. Enter the character set encoding to use for the data in the payload in the **Encoding** field.
9. Enter a string to use for padding the data in the payload in the **Padding** field.
10. Click **Next** to bring up the **Operations Editor** as shown in Figure 26 on page 64.
11. Click **Add** to add a new operation to the table.
12. Select the new operation from the table.
13. Click **Edit** to bring up the operation editing table shown in Figure 27 on page 65.
14. Enter a name for the operation in the **Name** field.
15. Select what type of operation you want to define from the **Style** drop-down list.
    - **REQUEST_RESPONSE** specifies that the operation will use two messages. One for the input message. The other message is for the output message.
    - **ONE_WAY** specifies that the operation will only have one message. The message can be either an input message or an output message.
16. Enter a discriminator string for the operation in the **Discriminator** field. See “fixed:operation” on page 171.
17. Select one of the messages from tree on the left to bring up the message editing table shown in Figure 28 on page 66.
18. Enter a name for the message in the **Name** field.

19. Select if this message is an input message or an output message from the **Type** drop-down list.
20. Enter the desired attributes for the message. See “fixed:body” on page 171.
21. Select the desired field type from the **Field Type To Add** drop-down list.
22. Click **Add** to add the new field to your message.
23. Select the field from the table.
24. Click **Edit** to define the new field.
    - **Fields** are atomic elements of a fixed binding. They are edited in the window, shown in Figure 29. For information on the values for a fixed field’s attributes see “fixed:field” on page 173.

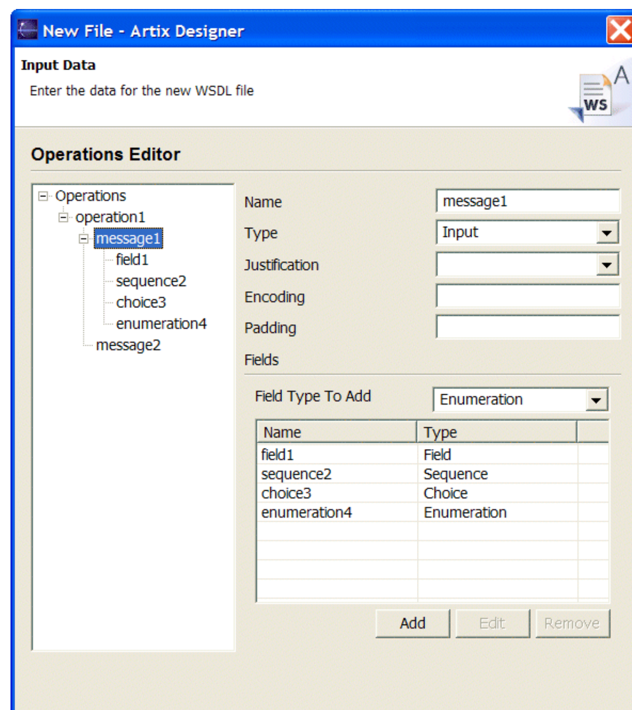


Figure 29: *Editing a Fixed Field*

- **Sequence** is a collection of fields that are mapped to a sequence complex type. You need to specify the attributes for the fixed sequence and then add at least one field to the sequence. For more information see "fixed:sequence" on page 179.
- **Choice** is a collection of fields that are mapped to a choice complex type. Each element of the complex type is mapped into a fixed case elements. You must enter a name for the choice and an optional discriminator string. You must also add at least one case element to the choice.

  To add a case to the choice click the **Add** button. When the new case appears in the table, select it so that you can edit its attributes. Cases are a collection of fields. For more information see "fixed:choice" on page 176.
- **Enumeration** is a special type of fixed field that allows you to specify the possible values that are valid for the field. They are mapped using the mapping shown in "Defining an enumeration in XML Schema" on page 119. You must add at least one possible enumeration value. Enumerated values are made up of two parts. **Value** is the value that is used in the logical description of the type. **Rendering** is the value that is placed on the wire when the data is written out. For more information see "fixed:field" on page 173.

25. Repeat steps **21** through **24** until all of the fields for the message are entered.
26. Repeat steps **17** through **25** for all messages in the operation.
27. Repeat steps **11** through **27** for all of the operations you want to define.
28. Click **Finish**.

Entering a tagged data set

If you select **Tagged** do the following:

1. Click **Next** to bring up the **Set Defaults** window, shown in Figure 30.

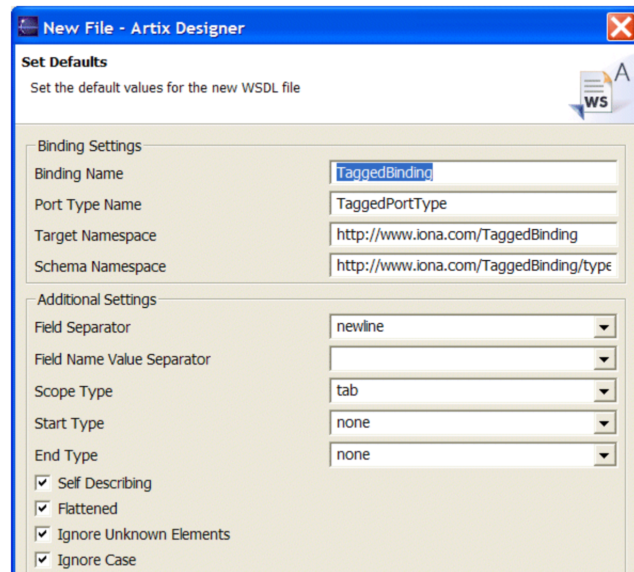


Figure 30: *Entering Defaults for a Contract from a Tagged Data Set*

2. Enter a name for the generated fixed binding in the **Binding Name** field.
3. Enter a name for the generated `portType` element in the **Port Type Name** field.
4. Enter a target namespace for the generated contract in the **Target Namespace** field.
5. Enter a target namespace for the generated `types` element in the **Schema Namespace** field.
6. Select a character to act as a field separator from the **Field Separator** drop-down list.
7. Select an optional character to separate the field names and the field values from the **Field Name Value Separator** drop-down list.

8. Select a scoping character from the **Scope Type** drop-down list.
9. Select a character to place at the beginning of messages from the **Start Type** drop-down list.
10. Select a character to place at the end of messages from the **End Type** drop-down list.
11. If you want the field names to be included in the messages, Select the **Self Describing** check box.
12. If you want data structures flattened when they are placed into messages, Select **Flattened**.
13. If you want Artix to ignore unknown fields when it receives messages, Select **Ignore Unknown Elements**.
14. If you want Artix to ignore the case of letters in field names, Select **Ignore Case**.
15. Click **Next** to bring up the **Operations Editor**.
16. Click **Add** to add a new operation to the table.
17. Select the new operation from the table.

18. Click **Edit** to bring up the operation editing table shown in Figure 31.

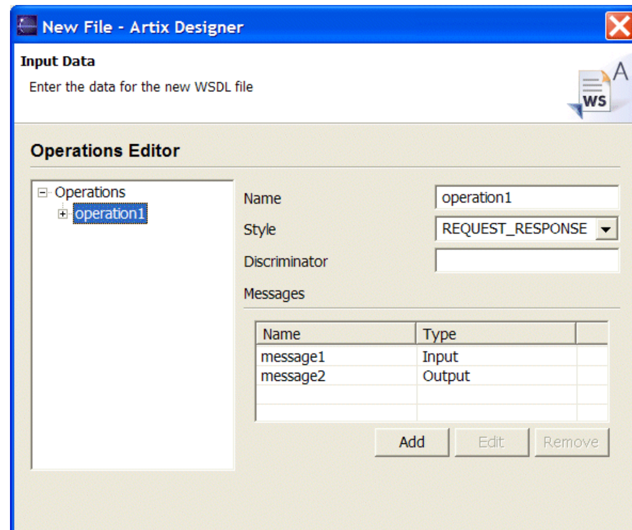


Figure 31: *Editing a Tagged Data Set Operation*

19. Enter a name for the operation in the **Name** field.
20. Select what type of operation you want to define from the **Style** drop-down list.
    - **REQUEST_RESPONSE** specifies that the operation will use two messages. One for the input message. The other message is for the output message.
    - **ONE_WAY** specifies that the operation will only have one message. The message can be either an input message or an output message.
21. Enter a discriminator string for the operation in the **Discriminator** field. See "tagged:operation" on page 193.

22. Select one of the messages from tree on the left to bring up the message editing table shown in Figure 32.

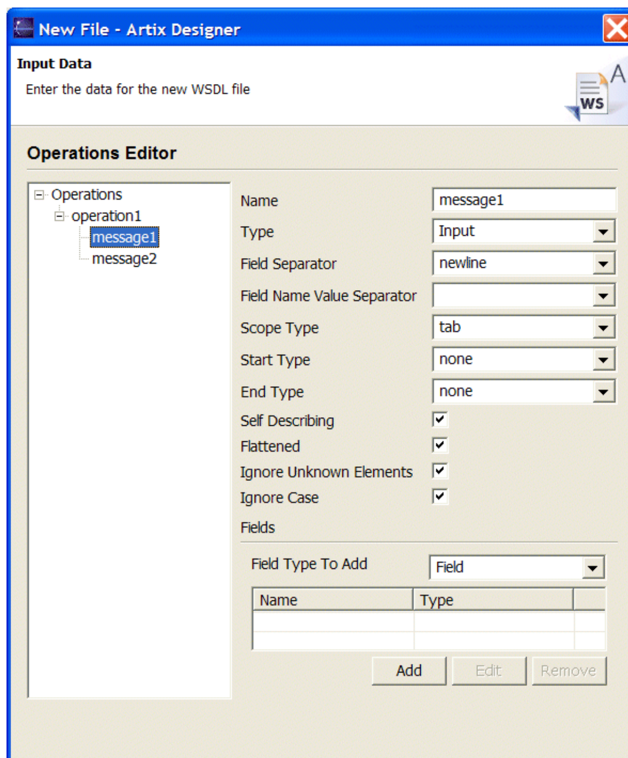


Figure 32: *Editing a Tagged Data Set Message*

23. Enter a name for the message in the **Name** field.
24. Select if this message is an input message or an output message from the **Type** drop-down list.
25. Enter the desired attributes for the message. See "tagged:body" on page 194.
26. Select the desired field type from the **Field Type To Add** drop-down list.
27. Click **Add** to add the new field to your message.

28. Select the field from the table.
29. Click **Edit** to define the new field.
    - **Fields** are atomic elements of a tagged binding. For information on the values for a tagged field's attributes see "tagged:field" on page 194.
    - **Sequence** is a collection of fields that are mapped to a sequence complex type. You need to specify the attributes for the tagged sequence and then add at least one field to the sequence. For more information see "tagged:sequence" on page 195.
    - **Choice** is a collection of fields that are mapped to a choice complex type. Each element of the complex type is mapped into a fixed case elements. You must enter a name for the choice and an optional discriminator string. You must also add at least one case element to the choice. To add a case to the choice, click the **Add** button. When the new case appears in the table, select it. Cases are a collection of fields. For more information see "tagged:choice" on page 197.
    - **Enumeration** is a special type of tagged field that allows you to specify the possible values that are valid for the field. They are mapped using the mapping shown in "Defining an enumeration in XML Schema" on page 119. You must add at least one possible enumeration value. For more information see "tagged:enumeration" on page 194.
30. Repeat steps **26** through **29** until all of the fields for the message are entered.
31. Repeat steps **22** through **30** for all messages in the operation.
32. Repeat steps **16** through **31** for all of the operations you want to define.
33. Click **Finish**.

Creating a Contract from an XML Schema Document

Overview

If you have an XML Schema document that defines all of the types you wish to use in a service, Artix provides a tool called `xsdtowsdl` that will import the schema document and produce a skeleton Artix contract.

The generated contract will contain only the type definitions from the imported schema document.

If you want to use the contract as a service definition, you will need to provide the remaining elements of the contract. However, if you simply want to import the generated contract into another contract, you do not need to do anything.

Usage

To generate an Artix contract from an XML Schema document use the following command:

```
xsdtowsdl [-t namespace] [-n name] [-d dir] [-o file] [-?] [-v]
          [-verbose] [-L file] [-q] [-h] [-V] xsdurl
```

The command has the following options:

`-t` *namespace*	Specifies the target namespace for the generated contract. The default is to use the Artix target namespace.
`-n` *name*	Specifies the name for the generated contract and is the value of the `name` attribute in the contract's root `definitions` element. The default is to use the schema document's file name.
`-d` *dir*	Specifies the output directory for the generated contract.
`-o` *file*	Specifies the filename for the generated contract. Defaults to the filename of the imported schema document. For example, if the imported schema document is stored in `maxwell.xsd` the resulting contract will be `maxwell.wsdl`.

`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Creating an XML Schema

Overview

XML Schema documents can be used to define types used in Artix contracts. Because XML Schema documents can be imported into any number of contracts, they are a convenient way to define types that are shared by multiple services.

When creating a new contract using the **XSD File** wizard you have two options:

- Creating an empty XML Schema document and the filling in the type definitions.
- Linking to an existing XML Schema document on a locally accessible file system.

Creating an empty XML Schema document

When you create an empty XML Schema document, the new document is placed under the project's file system. To create an empty XML Schema document in your project's workspace:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15 on page 35.
2. Select **XSD File** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **XSD File** window.
4. Select the folder where you want to store the XML Schema document.
5. Type the name of the file in the **File name** field.
6. Click **Finish**.

Linking to an existing XML Schema document

When you link to an existing XML Schema document, you create an entry in your project for the XML Schema document. The entry in your project is not a copy of the original file. It is a hard link to the original document and any changes made to the document are reflected in the original XML Schema document.

To link to an existing XML Schema document:

1. From the **File** menu, select **New|Other**.
2. Select **XSD File** from under the **IONA Artix Designer** folder.
3. Click **Next**.

4. Select the folder where you want to store the XML Schema document.
5. Type the name of the file in the **File name** field.
6. Click **Advanced** to expand the **XSD File** window.
7. Select the **Link to file in the file system** box.
8. Enter path name of the XML Schema document you want to import into the text box.
9. Click **Finish**.

Importing an XML Schema from a URL

Overview

If you want to import type definitions from an existing XML Schema document that is posted at a URL, you can import a read-only version of the XML Schema document into an Artix project.

Procedure

To import an XML Schema document from a URL:

1. From the **File** menu, select **New|Other** to open the **New** window, shown in Figure 15 on page 35.
2. Select **XSD From URL** from under the **IONA Artix Designer** folder.
3. Click **Next** to open the **XSD File** window.
4. Select the folder where you want to store the XML Schema document.
5. Type the name of the document in the **File name** field.
6. Type the URL in the **URL Path** field.
7. Click **Finish**.

Creating Access Control Lists

Overview

Artix enables you to create access control lists (ACLs) to determine which roles can access the different operations contained within a WSDL file's port types. You can define ACLs at the port type level and have them applied to all operations for that port type. Or you can define them at the operation level. For more information on using ACLs see the Artix Secutiry Guide.

Procedure

To create an access control list for a WSDL file:

1. Right-click the WSDL file.
2. Select **Artix|New Access Control List** from the pop-up menu to open the **Select ACL Role** window, shown in Figure 33.

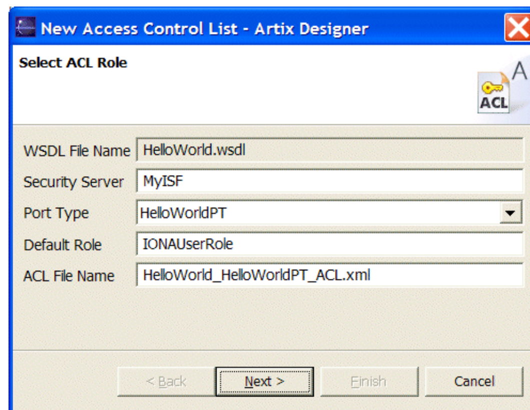


Figure 33: *Selecting Roles for an Access Control List*

3. Enter the fully scoped Artix domain name for the server that is doing the authentication in the **Security Server** field. See Deploying and Managing Artix Solutions for more information on Artix domain names.
4. Select the name of the `portType` element that defines the interface you want to secure from the **Port Type** drop-down list.
5. Enter a default security role in the **Default Role** field.

6. Enter a name for the generated ACL file in the **ACL File Name** field.
7. Click **Next** to bring up the Define ACL Operations window, shown in Figure 34.

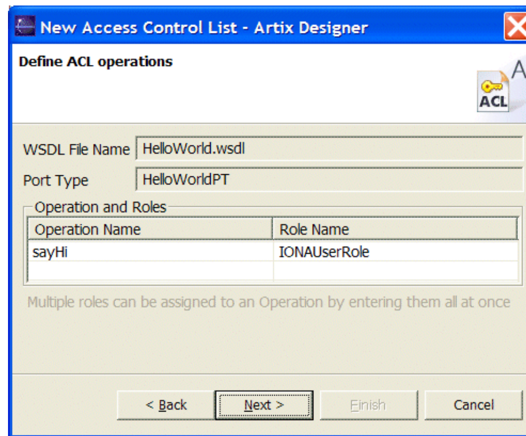


Figure 34: *Editing ACL Roles for Operations*

8. For each operation, select a enter a role into the **Role Name** field.

   You can enter data in the field by either:

   - Enter data into the field.
   - Select an entry from the drop-down list.
9. Click **Next** to review the contents of the generated access control list.
10. Click **Finish**.

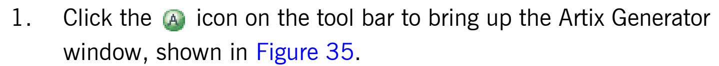
Working with Generation Profiles

Overview

Generation profiles define the type of generated files Artix creates for a project. They do not show up in a project's file tree. Instead they are managed by the Artix Generator.

Creating a generation profile

To create a generation profile:

1. Click the  icon on the tool bar to bring up the Artix Generator window, shown in Figure 35.

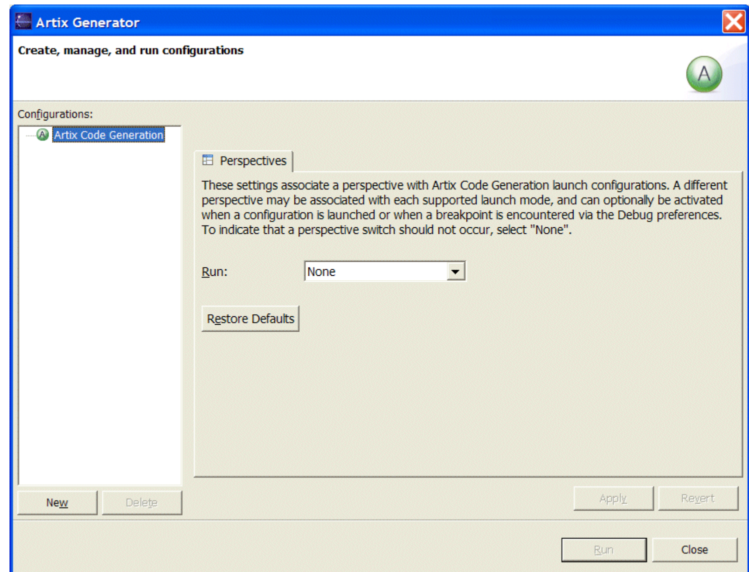


Figure 35: *The Artix Generator Window*

2. Click **New** to switch the window to editing mode
3. Enter a name for the generation profile in the **Name** field

4. In the General tabbed page, enter an optional description of the generation profile in the **Description** field.

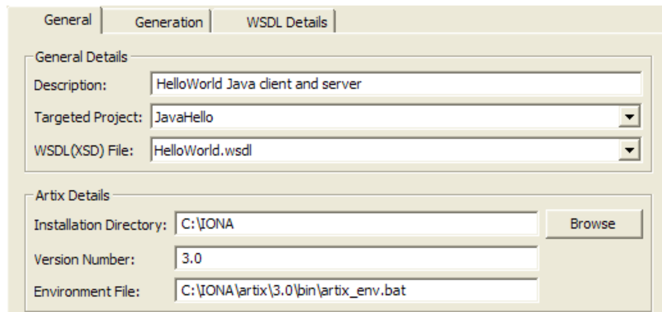


Figure 36: *The Artix Generator General Tabbed page*

5. From the **Targeted Project:** drop-down list, select the project for which this profile generates files.
6. From the **WSDL(XSD) File:** drop-down list, select the file from which this profile generates files.
7. Do not edit the values under **Artix Details**.
8. Click **Apply** to save your changes.

9. Click the **Generation** tab, shown in Figure 37, to edit the details of what this profile generates.

Figure 37: *The Artix Generator Generation Tabbed Page*

10. Select one of the generation profile types under **Generation Type**.

    The generation profile types are defined in Table 1 on page 32.
11. If you selected **Application**, select the type of application code you want to be generated under **Application Type**.
12. If you selected **Service plug-in** or **Application**, select your development language under **Development Language**.
13. Under **Optional Services**, select any of the Artix services you want to have included in the generated configuration domain.
14. Select the desired code generation settings.
15. Click **Apply** to save your changes.
16. If you are generating a **Container** or a **Switch** profile, skip to step **24**.

17. Click the **WSDL Details** tab, shown in Figure 38, to specify the services for which code will be generated.

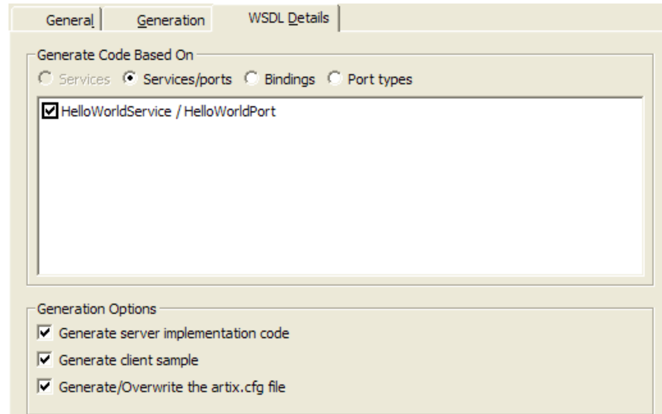
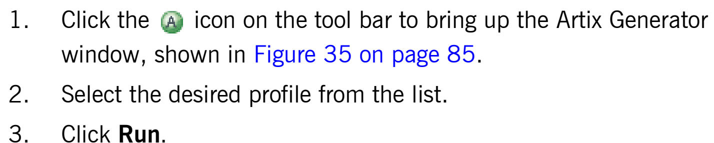


Figure 38: *The Artix Generator WSDL Details Tabbed Page*

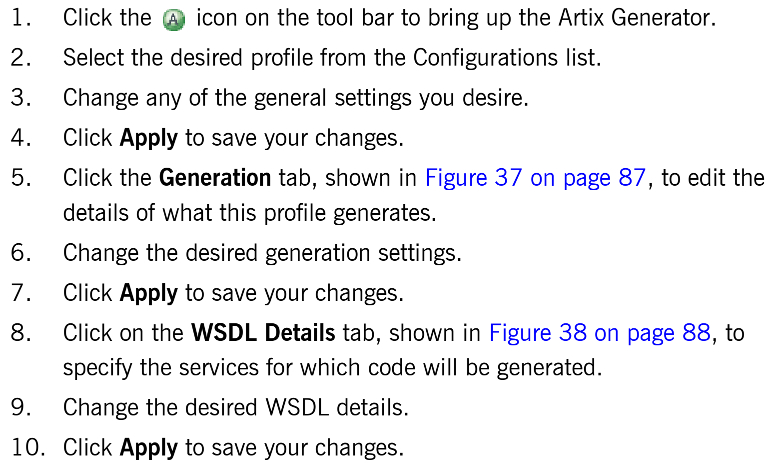
18. Select the part of the contract for which you want to generate code.
19. From the list select the name of the node for which code will be generated.
20. If you want starting point code for a server generated, click in the **Generate server implementation code** box.
21. If you want to generate a sample client, click in the **Generate client sample** box.
22. If you want to generate a new Artix configuration file every time you run the profile, click in the **Generate/Overwrite the artix.cfg file** box.
23. Click **Apply** to save your changes.
24. Click **Run** to execute the generation profile.

Running a generation profile



1. Click the icon on the tool bar to bring up the Artix Generator window, shown in Figure 35 on page 85.
2. Select the desired profile from the list.
3. Click **Run**.

Editing a generation profile

1. Click the  icon on the tool bar to bring up the Artix Generator.
2. Select the desired profile from the Configurations list.
3. Change any of the general settings you desire.
4. Click **Apply** to save your changes.
5. Click the **Generation** tab, shown in Figure 37 on page 87, to edit the details of what this profile generates.
6. Change the desired generation settings.
7. Click **Apply** to save your changes.
8. Click on the **WSDL Details** tab, shown in Figure 38 on page 88, to specify the services for which code will be generated.
9. Change the desired WSDL details.
10. Click **Apply** to save your changes.

Defining Data Types

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

In this chapter

This chapter discusses the following topics:

Introducing Data Types	page 92
Creating New Type Systems	page 93
Specifying a Type System in a Contract	page 95
XML Schema Simple Types	page 96
Defining Complex Data Types	page 98
Defining Elements	page 120

Introducing Data Types

Overview

When defining an interface in an Artix contract, the first thing you need to consider is the types of data that are used by the operation parameters of the interface. Artix uses XML Schema as its native type system.

XML Schema supports a number of simple types that do not require you to describe them in the contract. XML Schema also supports the definition of complex data types that are either a collection of typed elements or a derivative of a simple type.

In an Artix contract, complex type definitions are entered in the `type` element.

Creating New Type Systems

Overview

Depending on how you choose to create your WSDL, creating new type 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 a type system for your solution. It automatically generates the header sections of the type section of a contract and the wizards lead you through the steps to create different data types.

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 types used in an Artix contract involves seven steps:

1. Determine all the data types used in the interface described by the contract.
2. Create a `type` element in your contract.
3. Create a `schema` element, as a child of the `type` element, following the guidelines in "Specifying a Type System in a Contract" on page 95.
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 99.
5. For each array, define the data type using a `complexType` element. See "Defining Arrays" on page 107.
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 115.

7. For each enumerated type, define the data type using a `simpleType` element. See "Defining Enumerated Types" on page 117.
8. For each element, define it using an `element` element. See "Defining Elements" on page 120.

Specifying a Type System in a Contract

Overview

According to the WSDL specification, you can use any type system you like to define data types in WSDL. However, the W3C specification states XML Schema (XSD) is the preferred canonical type system for a WSDL document. Therefore, XSD is the intrinsic type system in Artix and is the only type system used by Artix Designer.

Specifying the type system

The first child element of the `types` element in a contract is the `schema` element. This element specifies the namespace for the types defined by the WSDL. It also defines the type system used to define the new types and any namespaces that are referenced in the type definitions.

Example 6 shows the standard `schema` element for an Artix contract. The attribute `targetNamespace` is where you specify the namespace under which your new data types are defined. The remaining entries are required. The first specifies that the types are defined using XML Schema. The second references a few special XML Schema types defined specifically for WSDL.

Example 6: *Artix Schema Element*

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

General guidelines

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.

XML Schema 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

XSD simple types are mainly placed in the `type` attribute of `element` elements used in defining sequences 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.

Supported XSD simple types

Artix supports the following XML Schema simple types:

- `xsd:string`
- `xsd:normalizedString`
- `xsd:int`
- `xsd:insignedInt`
- `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

XML Schema provides a flexible and powerful mechanism for building complex data types 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 99
Defining Arrays	page 107
Defining Types by Extension	page 109
Defining Types by Restriction	page 115
Defining Enumerated Types	page 117

Defining Data Structures

Overview

In XML Schema data structures 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 104.
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 105.

For example the structure shown in Example 7 would be defined in XML Schema as a `complexType` with two elements.

Example 7: *Simple Structure*

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

Example 8 shows one possible XML Schema mapping for `personalInfo`.

Example 8: *A Complex Type*

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

Using Artix Designer

To add a new data structure using theArtix Designer Diagram view:

1. Right-click the **Types** node to activate the pop-up menu.
2. Select **New Type** from the pop-up menu to bring up the **Select Source Resources** window, shown in Figure 39.

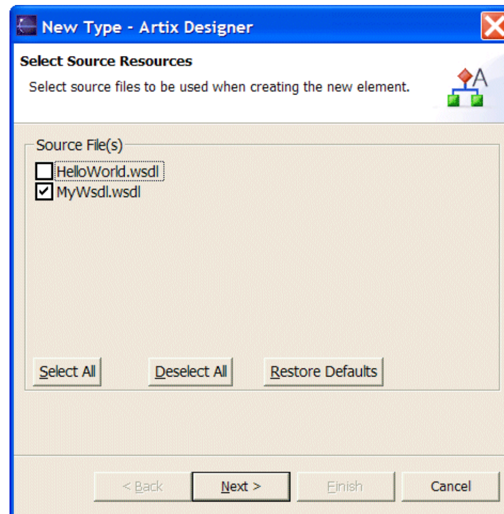


Figure 39: *Select Resource to which a New Type is Added*

3. Select at least one resource from the list to act as a source of predefined types.

   All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XML Schema types. The resources will also be imported to the target resource using WSDL `import` elements.

4. Click **Next** to bring up the **Define Type Properties** window, shown in Figure 40.

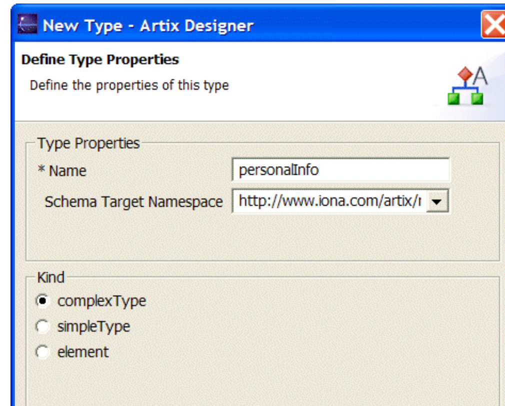


Figure 40: *Defining a Type's General Properties*

5. Enter a name for the new type in the **Name** field.
6. Enter the target namespace for the new type's XML Schema in the **Schema Target Namespace** field.

   You can either enter a new target namespace manually or, if your resource has multiple schema namespaces defined within it, you can select one of the existing namespaces from the drop-down list.
7. Under **Kind**, select **complexType**.

8. Click **Next** to bring up the **Define Complex Type Data** window, shown in Figure 41.

Figure 41: *Adding Elements to a Structure*

9. Select the type of structure you want to add from the **Group Type** drop-down list. See “complexType varieties” on page 104.
10. Select the type of the element you want to add to the structure from the **Type** drop-down list.
11. Enter a name for the new element in the **Name** field.
12. Select the **Nillable** check box if this element could potentially be omitted completely, or could pass an empty object across the wire.
13. To specify the minimum number of times the element must appear in the structure, enter a value in the **Min Occurrences** field. The default value is one.
14. To specify the maximum number of times the element can appear in the structure, enter a value in the **Max Occurrences** field. The default value is one.

15. If you want the element to be able to appear an unlimited number of times, select the **Unbounded** check box.
16. Click **Add** to save the element to the **Element List** table.
17. If you need to edit an element definition:
    i. Select the element from the **Element List** table.
    ii. The values for the element will populate the **Element Data** fields and the **Add** button will change to **Update**.
    iii. Make your changes and then click **Update**
18. Repeat steps **10** through **17** until you have finished adding elements to the structure.
19. Click **Next** to bring up the **Define Types Attributes** window, shown in Figure 42.

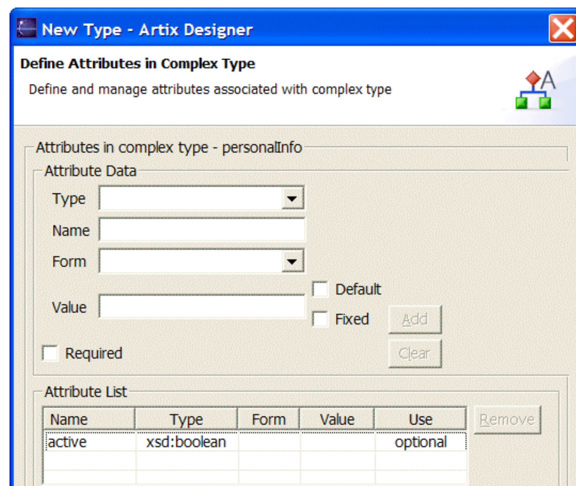


Figure 42: *Adding Attributes to a Structure*

20. Select the type of the attribute you want to add to the structure from the **Type** drop-down list.
21. Enter a name for the new attribute in the **Name** field.
22. If the attribute must be present in every instance of this type, select the **Required** check box.

23. Click **Add** to save the attribute to the **Attribute List** table.
24. If you need to edit an attribute definition:
    i. Select the attribute from the **Attribute List** table.
    ii. The values for the attribute will populate the **Attribute Data** fields and the **Add** button will change to **Update**.
    iii. Make your changes and then click **Update**
25. Repeat steps **20** through **24** until you have finished adding attributes to the structure.
26. Click **Finish** to add the structure.

---

complexType varieties

XML Schema 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 3 shows the elements used to define complex type behavior.

Table 3: *complexType Descriptor Elements*

Element	complexType Behavior
`sequence`	All the complex type's fields must be present and in the exact order they are specified in the type definition.
`all`	All the complex type's fields must be present but can be in any order.
`choice`	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 8 would generate a message containing two elements: `name` and `age`.

If the structure was defined as a choice, as shown in Example 9, it would generate a message with either a name element or an age element.

Example 9: *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: minOcurrs 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 10.

Example 10: *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 11. In this case `age` can be omitted and the data will still be valid.

Example 11: *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.

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 define a complex type with a single element with occurrence constraint placed on it. 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 `float`s you would use a complex type similar to the one shown in Example 12.

Example 12: *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 13.

Example 13: *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 14, 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 14: *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 15.

Example 15: *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 development languages, XML Schema 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 8 on page 99 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 `extenstion` element as they would be for a regular complex type.

For example, `alienInfo` would be defined as shown in Example 16.

Example 16: *Type Defined by Extension*

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

Defining by extension in Artix Designer

To define a complex type by extension in Artix Designer:

1. Right-click the **Types** node to activate the pop-up menu.
2. Select **New Type.** from the pop-up menu to bring up the **Select Source Resources** window, shown in Figure 39 on page 100.
3. Select at least one resource from the list to act as a source of predefined types. All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XML Schema types. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Define Type Properties** window, shown in Figure 40 on page 101.
5. Enter a name for the new type in the **Name** field.
6. Enter the target namespace for the new type's XML Schema in the **Schema Target Namespace** field.

   You can either enter a new target namespace manually or, if your resource has multiple schema namespaces defined within it, you can select one of the existing namespaces from the drop-down list.
7. Under **Kind**, select **complexType**.
8. Click **Next** to bring up the **Define Complex Type Data** window.

9. Select **complexContent** from the **Group Type** drop-down list to activate the **Content Base Type** drop-down list as shown in Figure 43.

Figure 43: *Defining the Base Type for Extension*

10. Select the type you want to extend from the **Content Base Type** drop-down list.
11. To add attributes to the extended type:
    i. Select a type for the new attribute from the **Type** drop-down list.
    ii. Enter a name for the new attribute in the **Name** field.
    iii. If the attribute is required, select the **Required** check box.
    iv. Click **Add** to save the new attribute to the **Attribute List** table.

    These attributes will be added inside of the `extension` element of the new type.
12. If you need to edit an attribute definition:
    i. Select the attribute from the **Attribute List** table.
    ii. The values for the attribute will populate the **Attribute Data** fields and the **Add** button will change to **Update**.

iii. Make your changes and then click **Update.**

13. Repeat steps 11 and 12 until you have finished adding attributes.
14. Click **Next** to bring up the **Define Complex Content Type Data** window, shown in Figure 44.

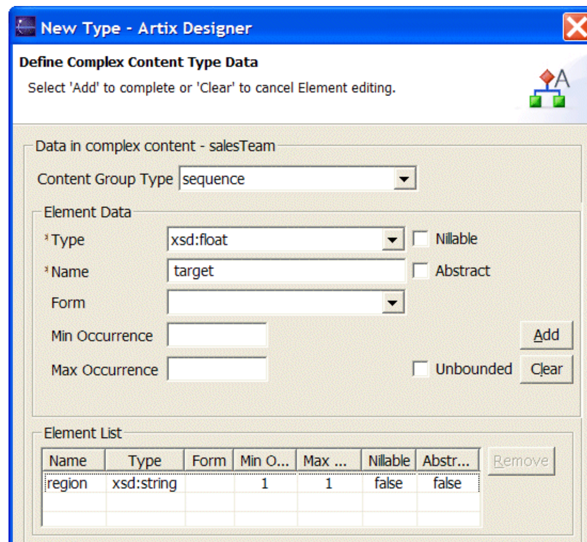


Figure 44: *Adding Elements to an Extended Type*

15. Select how you want the new elements in the extended type to be organized from the **Content Group Type** drop-down list, as follows:
    - **all**—all elements must be present, but the order does not matter.
    - **sequence**—all elements must be present and listed in the order specified.
    - **choice**—only one element can be present in an instance of the type.
16. To add elements to the extended type:

    i. Select a type for the new element from the **Type** drop-down list.

    ii. Enter a name for the new element in the **Name** field.

    iii. Enter the minimum number of times the element must appear in an instance of the type in the **Min Occurrence** field. Default is 1.

iv. Specify the maximum number of times the element can appear in an instance of the type in the **Max Occurrence** field. Default is 1.

v. If the element can appear an unlimited number of times, select the **Unbounded** check box.

vi. If the element is optional, select the **Nillable** check box.

vii. Click **Add** to save the new element to the **Element List** table.

17. If you need to edit an element definition:

    i. Select the element from the **Element List** table.

    ii. The values for the element will populate the **Element Data** fields and the **Add** button will change to **Update**.

    iii. Make your changes and then click **Update**

18. Repeat steps **16** and **17** until you have finished adding elements.

19. Click **Next** to bring up the **Define Attributes in Complex Type** window, shown in Figure 45.

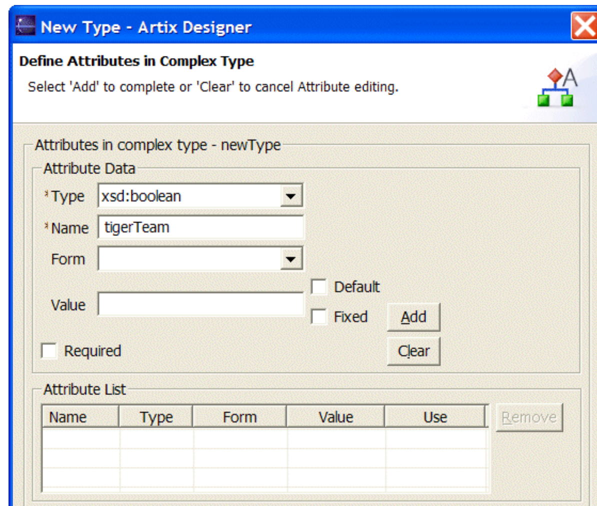


Figure 45: *Adding Global Attributes to an Extended Complex Type*

20. To add attributes to the new complex type:
    i. Select a type for the new attribute from the **Type** drop-down list.
    ii. Enter a name for the new attribute in the **Name** field.
    iii. If the attribute is required, select the **Required** check box.
    iv. Click **Add** to save the new attribute to the **Attribute List** table.

    These attributes will be added outside of the `extension` element of the new type.
21. If you need to edit an attribute definition:
    i. Select the element from the **Attribute List** table.
    ii. The values for the element will populate the **Attribute Data** fields and the **Add** button will change to **Update**.
    iii. Make your changes and then click **Update**.
22. Repeat steps **20** and **21** until you have finished adding elements.
23. Click **Finish**.

Defining Types by Restriction

Overview

XML Schema allows you to create new types by restricting the possible values of an XML Schema 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 115.
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 115.

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 XML Schema simple types.

For example, to define a new type by restricting the values of an `xsd:int` you would use a definition like Example 17.

Example 17: *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.

Designing a simple type with Artix Designer

The Designer has limited support for defining types by restriction. It will only allow you to specify the `maxLength` and `length` facets for `xsd:string`. It is more tailored to defining enumerations. See "Defining Enumerated Types" on page 117.

Example

Example 18 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 18: *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 XML Schema are a special case of definition by restriction. They are described by using the `enumeration` facet which is supported by all XML Schema 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 Artix Designer

To define a enumerated type from the Artix Designer Diagram view:

1. Right-click the **Types** node to activate the pop-up menu.
2. Select **New Type** from the pop-up menu to bring up the **Select Source Resources** window, shown in Example 39 on page 100.
3. Select at least one resource from the list to act as a source of predefined types.

   All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XML Schema types. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Define Type Properties** window, shown in Figure 40 on page 101.
5. Enter a name for the new type in the **Name** field.
6. Enter the target namespace for the new type's XML Schema in the **Schema Target Namespace** field.

   You can either enter a new target namespace manually or, if your resource has multiple schema namespaces defined within it, you can select one of the existing namespaces from the drop-down list.
7. Under **Kind**, select **simpleType**.

8. Click **Next** to bring up the **Define Type Data** window, shown in Figure 46.

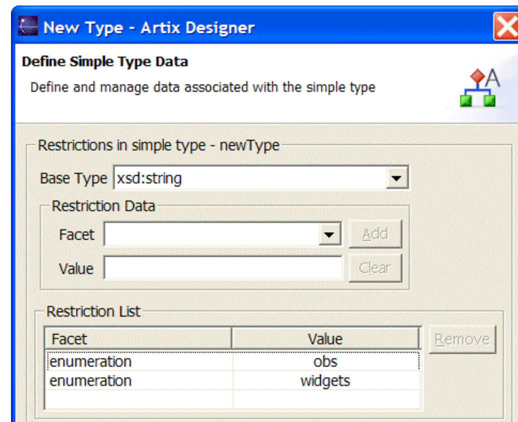


Figure 46: *Defining the Values for an Enumeration*

9. Chose a base type for the enumeration from the **Base Type** drop-down list.
10. Add the values for the enumeration:
    i. Select **enumeration** from the **Facet** drop-down list.
    ii. Enter a value in the **Value** field.
    iii. Click **Add** to enter the value into the **Restriction List** table.
    iv. Repeat until all of the values are added to the table.
11. If you need to edit an enumeration value:
    i. Select the element from the **Restriction List** table.
    ii. The values will populate the **Restriction Data** fields and the **Add** button will change to **Update**.
    iii. Make your changes and then click **Update**
    iv. Repeat until all values are edited.
12. Click **Finish**.

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in Example 19.

Example 19: *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 20, would be valid if it contained `<widgetSize>big</widgetSize>`, but not if it contained `<widgetSize>big,mungo</widgetSize>`.

Example 20: *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 XML Schema 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 ommited 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 an element using Artix Designer

To define an element from the diagram view of Artix Designer :

1. Right-click the **Types** node to activate the pop-up menu.
2. Select **New Type** from the pop-up menu to bring up the **Select Source Resources** window, shown in Figure 39.
3. Select at least one resource from the list to act as a source of predefined types.

   All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XML Schema types. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Define Type Properties** window, shown in Figure 40.

5. Enter a name for the new type in the **Name** field.
6. Enter the target namespace for the new type's XML Schema in the **Schema Target Namespace** field.

   You can either enter a new target namespace manually or, if your resource has multiple schema namespaces defined within it, you can select one of the existing namespaces from the drop-down list.
7. Under **Kind**, select **Element**.
8. Click **Next** to bring up the **Define Element Data** window, shown in Figure 47.

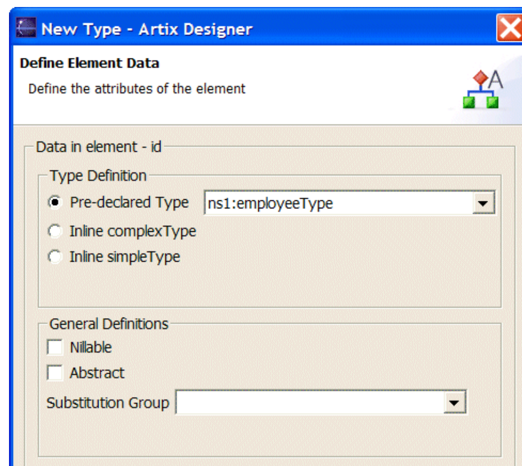


Figure 47: *Defining the Base Values for an Element Definition*

9. Select how you intend to define the type of the element:
   - Pre-declared Type allows you to choose a type you have already defined or one of the native types from the drop down list.
   - Inline complexType walks you through the process of defining a new complex type to define the data stored in the element. See "Defining Complex Data Types" on page 98.
   - Inline simpleType walks you through the process of defining a new enumeration to define the data stored in the element. See "Defining Enumerated Types" on page 117.

10. Select **Nillable** if this element could potentially be omitted completely, or could pass an empty object across the wire.
11. Select **Abstract** to define the element as an abstract head element. This means the element can never appear in a document produced using this schema.
12. If the element you are creating can replace another element in documents generated from this schema, select the replaceable element from the **Substitution Group** drop-down list. A substitution group allows you to build a collection of elements that can be specified using a generic element.
13. Click **Next**.
14. Click **Finish**.

CHAPTER 6

Defining Messages

You can define complex messages to pass between your services.

Overview

WSDL is designed to describe how data is passed over a network and because of this it describes data that is exchanged between two endpoints in terms of abstract messages described in `message` elements.

Each abstract message consists of one or more parts, defined in `part` elements. These abstract messages represent the parameters passed by the operations defined by the WSDL document and are mapped to concrete data formats in the WSDL document's `binding` elements.

Messages and parameter lists

For simplicity in describing the data consumed and provided by an endpoint, WSDL documents allow abstract operations to have only one input message, the representation of the operation's incoming parameter list, and one output message, the representation of the data returned by the operation. In the abstract message definition, you cannot directly describe a message that represents an operation's return value, therefore any return value must be included in the output message

Messages allow for concrete methods defined in programming languages like C++ to be mapped to abstract WSDL operations. Each message contains a number of `part` elements that represent one element in a parameter list. Therefore, all of the input parameters for a method call are defined in one message and all of the output parameters, including the operation's return value, would be mapped to another message.

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 21.

Example 21: *personalInfo lookup Method*

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

This method signature could be mapped to the WSDL fragment shown in Example 22.

Example 22: *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 WSDL document 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 elements of the abstract message. Each part is identified by a name and an attribute specifying its data type. The data type attributes are listed in Table 4

Table 4: *Part Data Type Attributes*

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

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 23.

Example 23: *Reused Part*

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

Defining messages with Artix Designer

To add a message to your contract using the Artix Designer Diagram view:

1. Right-click the **Message** node to activate the pop-up menu.
2. Select **New Message** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of data types.

   All of the types defined in the selected resources, as well as the native XML Schema types, will be made available for you to use in defining messages. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Define Message Properties** window, shown in Figure 48.

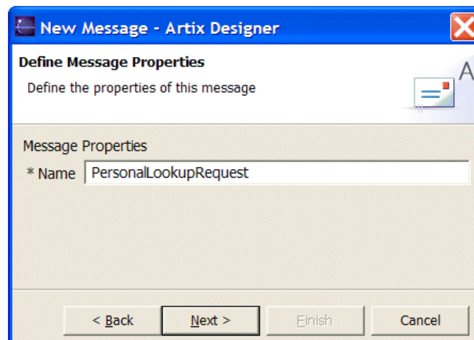


Figure 48: *Naming a Message*

5. Enter a name for your message in the **Name** field.
6. Click **Next** to bring up the **Define Message Parts** window, shown in Figure 49.

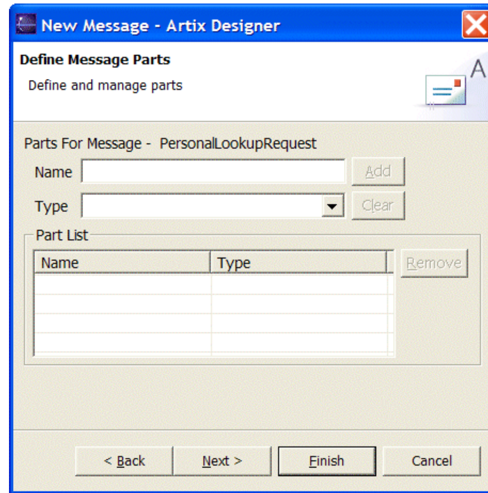


Figure 49: *Adding Parts to a Message*

7. Enter a name for the message part in the **Name** field.
8. Select a data type from the **Type** drop-down list.
9. Click **Add** to save the new part to the **Part List** table.
10. If you need to edit a part definition:
    i. Select the part from the **Part List** table.
    ii. The values for the part will populate the fields at the top of the window and the **Add** button will change to **Update**.
    iii. Make your changes and then click **Update**
11. Repeat steps **7** through **10** until you have finished adding parts to the message.
12. Click **Finish**.

Defining Your Interfaces

In WSDL documents interfaces are defined using the portType element.

Overview

Interfaces are defined using the WSDL `portType` element. Like an interface, the `portType` is a collection of operations that define the input, output, and fault messages used by the service implementing the interface to complete the transaction the operation describes. The difference is that the operations in a port type are built up using messages that are defined outside of the port type instead of parameter lists defined as part of the operation itself.

Defining an interface in an Artix contract entails following:

1. Creating a `portType` element to contain the interface definition and give it a unique name. See "Port types" on page 128.
2. Creating an `operation` element for each operation defined in the interface. See "Operations" on page 128.
3. For each operation, specifying the messages used represent the operation's parameter list, return type, and exceptions. See "Operation messages" on page 128.

The Artix Designer automates the process of creating new port types. See "Defining an interface with Artix Designer" on page 129.

Port types

A portType element is the root element in an interface definition and many Web service implementations, including Artix, map port types directly to generated implementation objects. In addition, the portType element is the abstract unit of a WSDL document that is mapped into a concrete binding to form the complete description of what is offered over a port.

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

Operations, described in operation elements in a WSDL document are an abstract description of an 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

Operations are made up of a set of elements representing the messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in Table 5.

Table 5: *Operation Message Elements*

Element	Description
input	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.
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 are defined by two attributes listed in Table 6.

Table 6: *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 `message` attribute must correspond to the `name` 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 in 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 "Binding Interfaces to a Payload Format" on page 135.

Defining an interface with Artix Designer

The add a new interface to your contract from the Artix Designer Diagram view:

1. Right-click the **Port Types** node to activate the pop-up menu.
2. Select **New Port Type** from the pop-up menu to bring up the **Select Source Resources** window.

3. Select at least one resource from the list to act as a source of messages.

   All of the messages defined in the selected resources will be made available for you to use in defining the interface's operations. The resources will also be imported to the target resource using WSDL `import` elements.

4. Click **Next** to bring up the **Define Port Type Properties** window, shown in Figure 50.

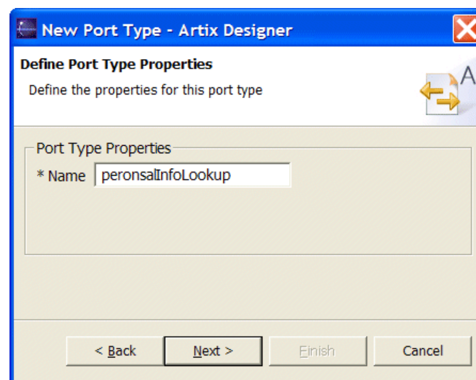


Figure 50: *Naming a Port Type*

5. Enter a name for the new interface in the **Name** field.

6. Click **Next** to bring up the **Define Port Type Operations** window, shown in Figure 51.

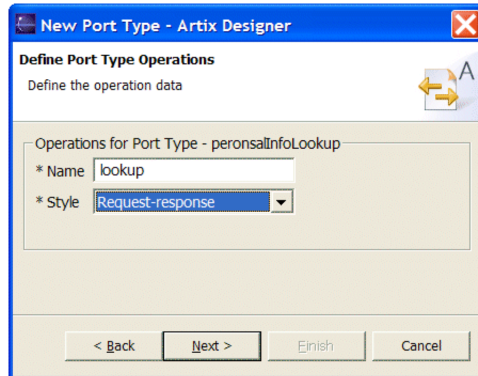


Figure 51: *Adding an Operation to a New Port Type*

7. Enter a name for the new operation in the **Name** field.
8. Select an operation style from the **Style** drop-down list.

   Operations can have one of the following styles:

   - **One-way** operations have only an input message. They cannot return any data to the client.
   - **Request-response** operations have an input message, an output message, and any number of optional fault messages.

9. Click **Next** to open the **Define Operation Messages** window, shown in Figure 52.

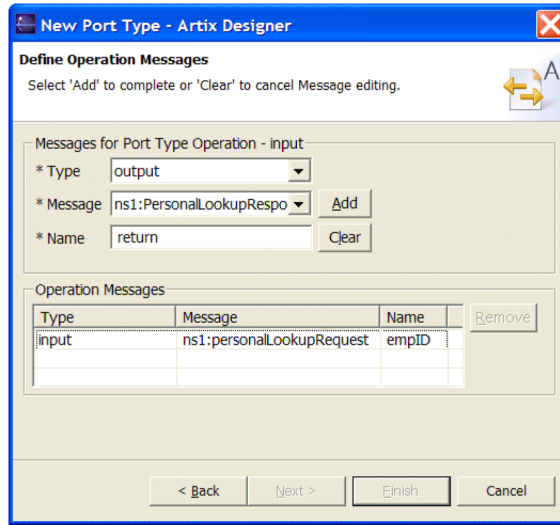


Figure 52: *Defining the Messages in an Operation*

10. Select a message type for the new operation message from the **Type** drop-down list.

    Operation messages can be of one of the following types:

    - **input** messages represent data that a client send to the server.
    - **output** messages represent data that a service returns to a client.
    - **fault** messages represent data that a service returns to a client in the event that an error occurred while processing the request.

    Note: If your operation is oneway, input will be the only message type available.

11. Select the global message that defines the data passed by this operation message from the **Message** drop-down list.
12. Enter a name for the operation message in the **Name** field.
13. Click **Add** to add the message to the **Operation Messages** table.

14. If you need to edit an operation message:
    i. Select the part from the **Operation Messages** table.
    ii. The values for the operation message will populate the fields under **Messages for Port Type Operation** and the **Add** button will change to **Update**.
    iii. Make your changes and then click **Update**
15. Repeat steps **10** through **14** until all of the operational messages have been specified.
16. Click **Finish** to write the changes to your resource.

If you want to add more than one operation to your interface, you can add them using the **New Operation** pop-up menu option. For details on adding a new operation to an interfaces see "Adding a new operation" on page 359.

Example

For example, you might have an interface similar to the one shown in Example 24.

Example 24: *personalInfo lookup interface*

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

This interface could be mapped to the port type in Example 25.

Example 25: *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/>
```

Example 25: *personalInfo lookup port type*

```
<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.

CHAPTER 8

Binding Interfaces to a Payload Format

You can bind your interfaces to a number of payload formats in Artix.

In this chapter

This chapter discusses the following topics:

Introducing Bindings	page 136
Adding a SOAP Binding	page 137
Adding a CORBA Binding	page 152
Adding an FML Binding	page 159
Adding a Fixed Binding	page 166
Adding a Tagged Binding	page 186
Adding a TibrvMsg Binding	page 203
Adding a Pure XML Binding	page 237
Adding a G2++ Binding	page 243

Introducing Bindings

Overview

To define an endpoint that corresponds to a running service, port types are mapped to bindings that describe how the abstract messages used by the interface's operations map to the data format used on the wire. These bindings are described in `binding` elements. A binding can map to only one port type, but a port type can be mapped to any number of bindings.

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.

Adding a SOAP Binding

Overview

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

For more information

For more detailed information on the SOAP binding and the specifics of the elements used in defining it see "SOAP Binding Extensions" on page 391.

In this section

This section discusses the following topics:

Adding a Default SOAP Binding	page 138
Adding SOAP Headers to a SOAP Binding	page 142
Sending Data Using SOAP with Attachments	page 148

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 as described in "Using Artix Designer" on page 138. The second is the command line tool `wsdltosoap` as described in "Using wsdltosoap" on page 139. The third is to use the **SOAP Enable** option as described in "Web Service Enabling a Service" on page 346.

For information on the elements used to define a SOAP binding see "SOAP Binding Extensions" on page 391.

Using Artix Designer

To add a SOAP binding from the Artix Designer Diagram view:

1. Right-click the **Bindings** node to activate the pop-up window.
2. Select **New Binding** to bring up the **Select Source Resources** window.
3. Select at least one contract from the list to act as a source for interfaces. All of the interfaces in the selected contracts will be made available to you later. The contracts will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Select Binding Type** window, shown in Figure 53.

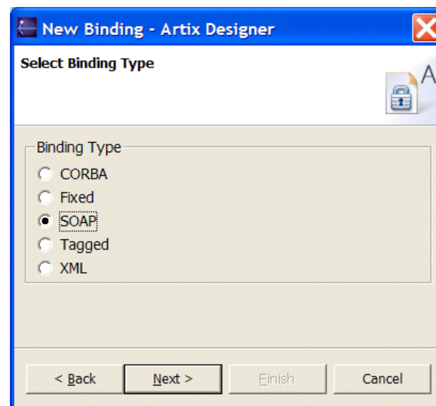


Figure 53: *Select the Type of Binding to Use*

5. Select **SOAP**.
6. Click **Next** to bring up the **Set Binding Defaults** window, shown in Figure 54.

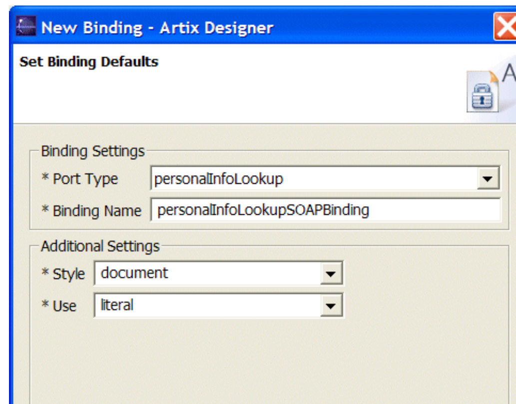


Figure 54: *Setting the Defaults for a SOAP Binding*

7. Select the interface which is being mapped to this SOAP binding from the **Port Type** drop-down list.
8. Enter a name for the binding in the **Name** field.
9. Select a style for the SOAP elements from the **Style** drop-down list.
10. Select a value for the SOAP `use` attribute from the **Use** drop-down list.
11. Click **Finish**.

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}]
           [-q] - [h] [-V]
```

The command has the following options:

`-i portType`	Specifies the name of the port type being mapped to a SOAP binding.

`-n` *namespace*	Specifies the namespace to use for the SOAP binding.
`-b` *binding*	Specifies the name for the generated SOAP binding. Defaults to *portType*`Binding`.
`-d` *dir*	Specifies the directory into which the new WSDL file is written.
`-o` *file*	Specifies the name of the generated WSDL file. Defaults to *wsdl_file*`-soap.wsdl`.
`-style`	Specifies the encoding style to use in the SOAP binding. Defaults to `document`.
`-use`	Specifies how the data is encoded. Default is `literal`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

`wsdltosoap` does not support thegeneration 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 26.

Example 26: *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>
```

Example 26: *Ordering System Interface*

```
<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 27.

Example 27: *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 28. 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 28: *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 headers using parts of the parent message, Artix automatically fills in the SOAP headers for you.

Example

Example 29 shows a modified version of the orderWidgets service shown in Example 26. 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 29: *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 29: *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="xsd1:keyElem"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="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>
```

Example 29: *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="xsd1:keyElem"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="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>
```

Example 29: *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 29 so that the header value was a part of the input and output messages as shown in Example 30. 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 30: *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 30: *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 `defintion` element to set this up is shown in Example 31.

Example 31: *MIM 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:multipartReleated` 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:multipartReleated` 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:mulipartRelated element as the first child element of the enclosing message.
2. Add a mime:part child element to the mime:multipartRelated element and set its name attribute to a unique string.
3. Add a soap:body element as the child of the mime:part element and set its attributes appropriately.

   If the contract had a default SOAP binding, you can copy the soap:body element from the corresponding message from the default binding into the MIME multipart message.
4. Add another mime:part child element to the mime:multipartReleated element and set its name attribute to a unique string.
5. Add a mime:content child element to the mime:part element to describe the contents of this part of the message.

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

   - 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 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 32 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 32: *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 32: *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>
```

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>` extension 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 "CORBA Type Mapping" on page 403.

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. See "Using Artix Designer" on page 153.

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 154.

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 155.

The fourth method CORBA enables your interface definition. It asks for basic information about the IIOP address for the new service and the interface for which the CORBA information will be generated. It then adds a CORBA binding and fully defined CORBA port to your contract. For more information see "CORBA Enabling a Service" on page 348.

Using Artix Designer

To add a CORBA binding to a contract:

1. Right-click the **Bindings** node to activate the pop-up window.
2. Select **New Binding** to bring up the **Select Source Resources** window.
3. Select at least one contract from the list to act as a source for interfaces.

   All of the interfaces in the selected contracts will be made available to you later. The contracts will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Select Binding Type** window, shown in Figure 53 on page 138.
5. Select **CORBA**.
6. Click **Next** to bring up the **Set Binding Defaults** window, shown in Figure 55.

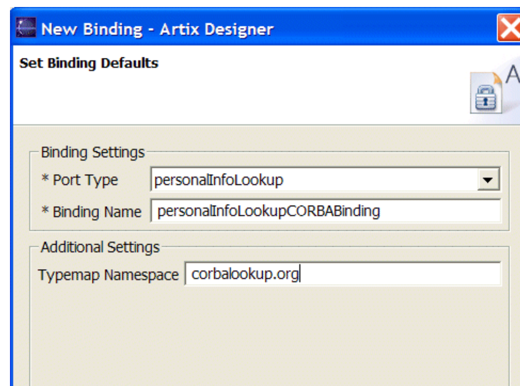


Figure 55: *Setting CORBA Binding Defaults*

7. Select the interface to be mapped to the CORBA binding from the **Port Type** drop-down list.
8. Enter a name for the new binding in the **Name** field.
9. Enter a namespace to use for the generated CORBA typemap into the **Typemap Namespace** field.
10. Click **Finish**.

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]
            [-n namespace] [-L file] [-q] [-h] [-V]  wsdl_file
```

The command has the following options:

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

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.iona.com/bindings/corba`. To use the CORBA extensions you will need to include the following in the `definitions` tag of your contract:

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

Mapping to the binding

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

corba:binding indicates that the binding is a CORBA binding. This element has one required attribute: `repositoryID`. `repositoryID` 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. The parameter order defined in the binding must match the order specified in the IDL definition of the operation. <corba:param> has the following required attributes:

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

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

corba:raises 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 33.

Example 33: *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 34.

Example 34: *personalInfoLookup CORBA Binding*

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

Adding an FML Binding

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 run time.
- 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 `fldid`s at run time.

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 7.

Table 7: *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`

Table 7: *FML Type Support (Continued)*

XML Schema Type	FML Type
`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 160.
2. Enter the FML binding information using Artix WSDL extensors as described in "Adding the FML binding" on page 164.

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 159.
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. We recommended, therefore, that you group elements into complex types only if they appear together in all exposed Tuxedo services.

For example, you may have 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 35.

Example 35: *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 36. 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 36: *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 employees ID number.

Because the employee data is always returned as a unit you can group it into a complex type as show in Example 37.

Example 37: *Logical description of personneInfo FML buffer*

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

The interface for your Tuxedo application would be mapped to a `portType` similar to Example 38.

Example 38: *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 38: *personnelInfo Lookup Interface*

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

Flattened XML and FML

While XML Schema allows you to create structured data that is organized in multiple layers, FML data is 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 37 on page 162, would be equivalent to the message definition in Example 39 when processed by the FML binding.

Example 39: *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.iona.com/transports/tuxedo"
```

2. Create a new `binding` element in your contract to define the FML buffer's binding.
3. Add an `tuxedo:binding` element to identify that this binding defines an FML buffer.
4. Add an `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 an `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 an `tuxedo:operation` element to the `operation` element.

For example, the binding for the `personalInfo` FML buffer, defined in Example 35 on page 161, will be similar to the binding shown in Example 40.

Example 40: *personalInfo FML binding*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="personalInfoService" targetNamespace="http://info.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo">
...
  <binding name="personelInfoFMLBinding" 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>
```

Adding a Fixed Binding

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 as shown in "Creating a Contract from a COBOL Copybook" on page 62.
- 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 as shown in "Creating a Contract from a Data Set" on page 69.
- 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 as shown in "Using Artix Designer" on page 166.
- You can enter the binding information using any text editor or XML editor as described in "Hand editing" on page 170.

Using Artix Designer

To add a fixed binding from the Artix Designer Diagram view:

1. Right-click the **Bindings** node to activate the pop-up window.
2. Select **New Binding** to bring up the **Select Source Resources** window.
3. Select at least one contract from the list to act as a source for interfaces.

   All of the interfaces in the selected contracts will be made available to you later. The contracts will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Select Binding Type** window.
5. Select **Fixed**.

6. Click **Next** to bring up the **Set Binding Defaults** window, shown in Figure 56.

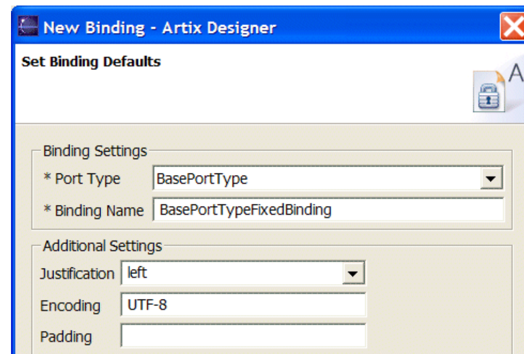


Figure 56: *Setting the Default Values for a Fixed Binding*

7. Select the interface mapped to this binding from the **Port Type** drop-down list.
8. Enter a name for the binding in the **Binding Name** field.
9. Select how you want the data in the payload to be justified from the Justification drop-down list.
10. Enter the character set encoding to use for the data in the payload in the **Encoding** field.
11. Enter a string to use for padding the data in the payload in the **Padding** field.

12. Click **Next** to bring up the **Edit Binding** window, shown in Figure 57.

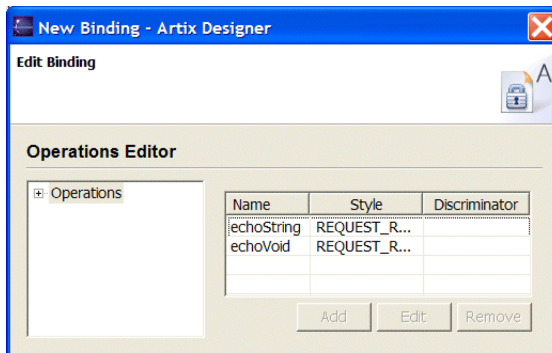


Figure 57: *Editing the Fixed Binding Settings*

13. Expand the Operations node to list the operations in the interface.
14. Select an operation from the tree to display its attributes, shown in Figure 58.

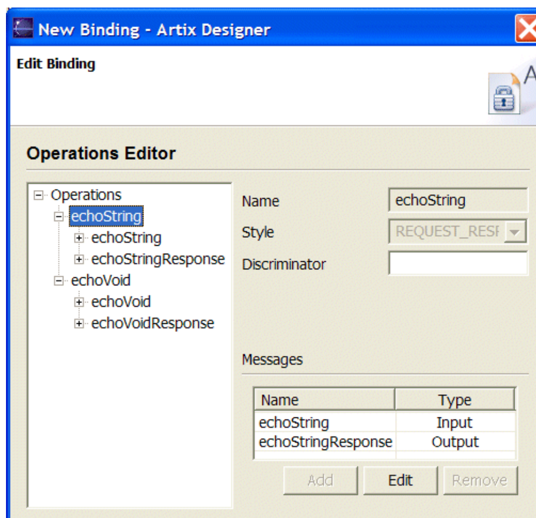


Figure 58: *Editing an Operation's Fixed Binding Settings*

15. Enter a discriminator string for the operation in the **Discriminator** field. See "fixed:operation" on page 171.
16. Select one of the operation's messages from the **Messages** table.
17. Click **Edit** to display the message attributes, shown in Figure 59.

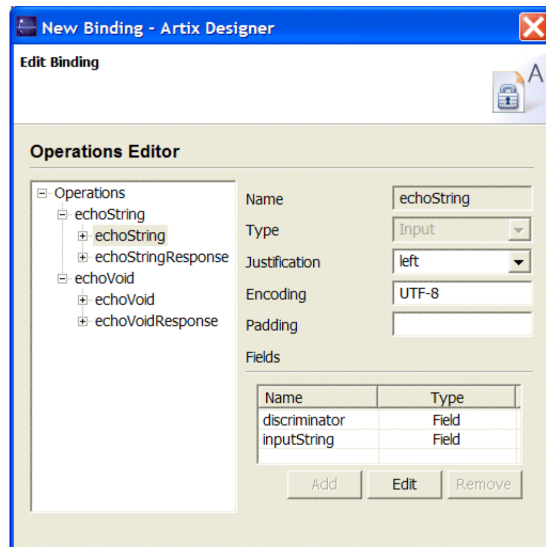


Figure 59: *Editing a Message's Fixed Binding Settings*

18. Edit the attributes for the message. See "fixed:body" on page 171.
19. Repeat step **18** for each message in the operation.
20. Repeat from step **14** until you have edited all the operations in the interface.
21. Click **Finish**.

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 170.
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 171.
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 171.
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 171.

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 8.

Table 8: *Attributes for fixed:binding*

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

The settings for the attributes on 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 "XML Schema Simple Types" on page 96.
    - `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 99.
    - `fixed:choice` maps message parts defined using a choice complex type. See "Defining Data Structures" on page 99.
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 41 would be mapped to a `fixed:field` similar to Example 42.

   Example 41: *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 42: *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 43.

Example 43: *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 44. This provides Artix with the meta-data needed to properly handle the data.

Example 44: *Mapping Numerical Data to a Fixed Binding*

```
<fixed:flield 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 45.

Example 45: *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 46.

Example 46: *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 45 on page 174, were mapped to the fixed field shown in Example 47, the actual message returned from the binding would always have the date 11/11/2112.

   Example 47: *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 117, 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 48.

   Example 48: *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 48 could be mapped to the fixed field shown in Example 49. Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to `OrangeTango`.

Example 49: *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 50. 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 50 must be two-digit integers.

Example 50: *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 51 shows an Artix contract fragment mapping a choice complex type to a fixed record length message.

Example 51: *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 51: *Mapping a Union to a Fixed Record Length Message*

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

fixed:sequence

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

1. Add a `fixed:sequnce` 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 99, 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 52. 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 52: *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 53 shows an Artix contract fragment for such a mapping.

Example 53: *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 53: *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 54 shows an example of an Artix contract containing a fixed record length message binding.

Example 54: *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 54: *Fixed Record Length Message Binding (Continued)*

```
<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 54: *Fixed Record Length Message Binding (Continued)*

```
<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 54: *Fixed Record Length Message Binding (Continued)*

```
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <fixed:binding/>
    <operation name="placeWidgetOrder">
      <fixed:operation discriminator="widgetDisc"/>
      <input name="widgetOrder">
        <fixed:body>
          <fixed:sequence name="widgetOrderForm">
            <fixed:field name="amount" format="###"/>
            <fixed:field name="order_date" format="MM/DD/YYYY"/>
            <fixed:field name="type" size="2">
              <fixed:enumeration value="big" fixedValue="bg"/>
              <fixed:enumeration value="large" fixedValue="lg"/>
              <fixed:enumeration value="mungo" fixedValue="mg"/>
              <fixed:enumeration value="gargantuan" fixedValue="gg"/>
            </fixed:field>
            <fixed:sequence name="shippingAddress">
              <fixed:field name="name" size="30"/>
              <fixed:field name="street1" size="100"/>
              <fixed:field name="street2" size="100"/>
              <fixed:field name="city" size="20"/>
              <fixed:field name="state" size="2"/>
              <fixed:field name="zip" size="5"/>
            </fixed:sequence>
          </fixed:sequence>
        </fixed:body>
      </input>
```

Example 54: *Fixed Record Length Message Binding (Continued)*

```
        <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>
```

Adding a Tagged Binding

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 a 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. See "Creating a Contract from a Data Set" on page 69.
- The Artix Designer can create a tagged binding for an existing interface automatically. See "Using the Artix Desinger" on page 186.
- You can enter the binding information using any text editor or XML editor. See "Hand editing" on page 191.

Using the Artix Desinger

To add a tagged binding from the Artix Designer Diagram view:

1. Right-click the **Bindings** node to activate the pop-up window.
2. Select **New Binding** to bring up the **Select Source Resources** window.
3. Select at least one contract from the list to act as a source for interfaces.

   All of the interfaces in the selected contracts will be made available to you later. The contracts will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to display the **Select Binding Type** window, shown in Figure 53 on page 138
5. Select **Tagged**.

6. Click **Next** to bring up the **Set Binding Defaults** window, shown in Figure 60.

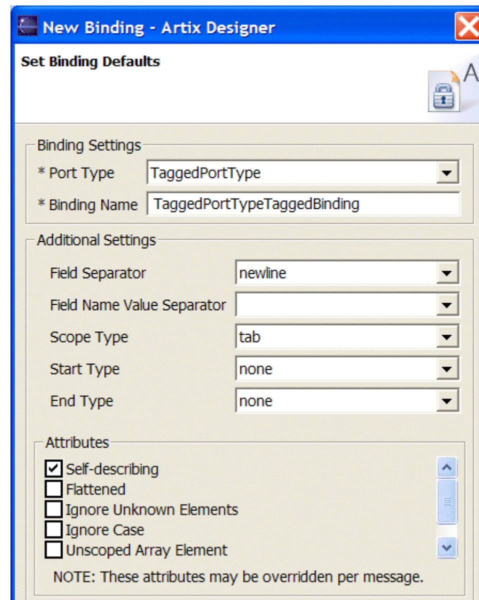


Figure 60: *Setting the Default Values for a Tagged Binding*

7. Select the interface to mapped to the binding from the **Port Type** drop-down list.
8. Enter a name for the binding in the **Binding Name** field.
9. Select the character to use a separator between data fields from the **Field Separator** drop-down list.
10. Select the character that will separate field names and their associated data values from the **Field Name Value Separator** drop-down list.
11. Select a character to use in scoping the data from the **Scope Type** drop-down list.
12. Select a character to signify the beginning of a message from the **Start Type** drop-down list.

13. Select a character to signify the end of a message from the **End Type** drop-down list.
14. Under **Attributes**, select the `true`/`false` attributes to activate for the tagged binding. See "tagged:binding" on page 192.
15. Click **Next** to bring up the **Edit Binding** window, shown in Figure 61.

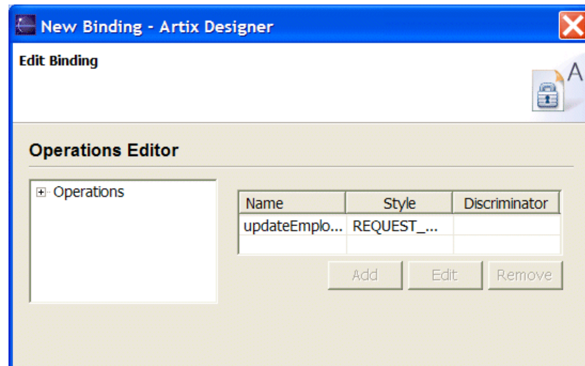


Figure 61: *Editing the Operations in the Fixed Binding*

16. Select one of the operations from the tree on the left to display its attributes, shown in Figure 62.

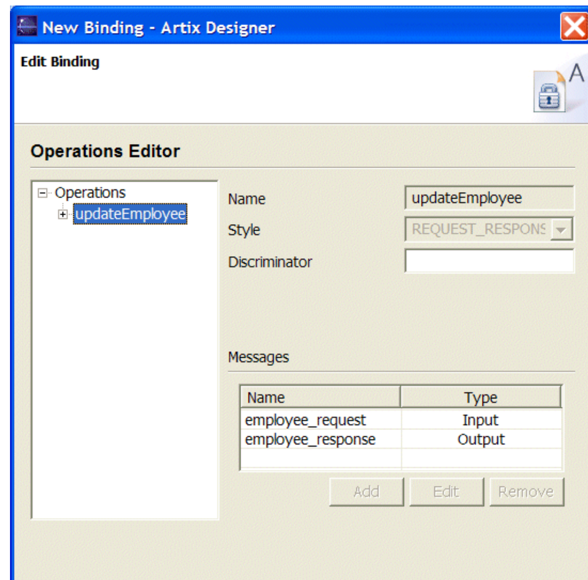


Figure 62: *Editing the Tagged Binding Settings for an Operation*

17. Specify a discriminator for the operation in the **Discriminator** field. See "tagged:operation" on page 193.
18. Select one of the operation's messages from the **Messages** table.

19. Click **Edit** to bring up the message editor shown in Figure 63.

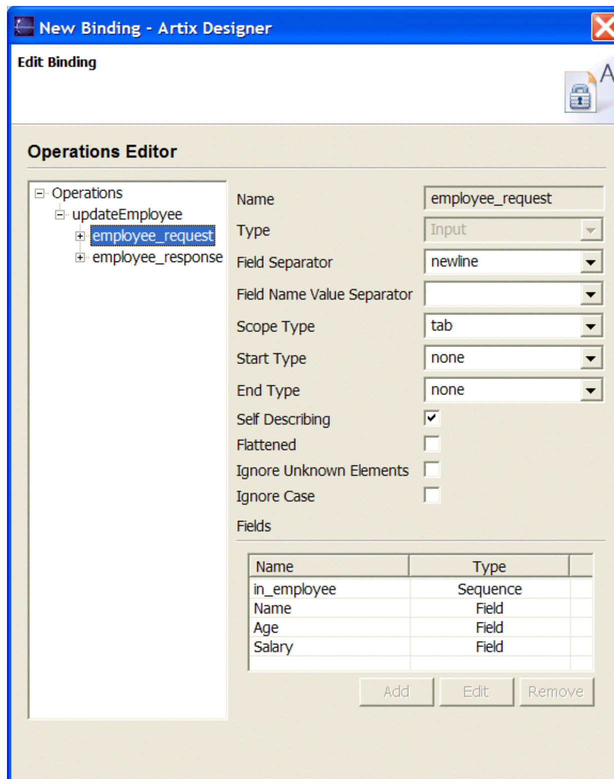


Figure 63: *Editing a Message's Tagged Binding Settings*

20. Edit the attributes for the message. See "tagged:body" on page 194.
21. Repeat step **20** for each message in the operation.
22. Repeat from step **16** until all operation in the interface have been edited.
23. Click **Finish**.

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 191.
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 9.

Table 9: *Attributes for tagged:binding*

Attribute	Purpose
`selfDescribing`	Required attribute specifying if the message data on the wire includes the field names. Valid values are `true` or `false`. If this attribute is set to `false`, the setting for `fieldNameValueSeparator` is ignored.
`fieldSeparator`	Required attribute that specifies the delimiter the message uses to separate fields. Supported values are `newline`(\n), `comma`(,), `semicolon`(;), and `pipe`(|).
`fieldNameValueSeparator`	Specifies the delimiter used to separate field names from field values in self-describing messages. Supported vales are: `equals`(=), `tab`(\t), and `colon`(:).
`scopeType`	Specifies the scope identifier for complex messages. Supported values are `tab`(\t), `curlybrace`({*data*}), and `none`. The default is `tab`.
`flattened`	Specifies if data structures are flattened when they are put on the wire. If `selfDescribing` is `false`, then this attribute is automatically set to `true`.
`messageStart`	Specifies a special token at the start of a message. It is used when messages that require a special character at the start of a the data sequence. Currently the only supported value is `star`(*).
`messageEnd`	Specifies a special token at the end of a message. Supported values are `newline`(\n) and `percent`(%).

Table 9: *Attributes for tagged:binding (Continued)*

Attribute	Purpose
`unscopedArrayElement`	Specifies if array elements need to be scoped as children of the array. If set to `true` arrays take the form `echoArray{myArray=2;item=abc;item=def}`. If set to `false` arrays take the form `echoArray{myArray=2;{0=abc;1=def;}}`. Default is `false`.
`ignoreUnknownElements`	Specifies if Artix ignores undefined element in the message payload. Default is `false`.
`ignoreCase`	Specifies if Artix ignores the case with element names in the message payload. Default is `false`.

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

tagged:operation

`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 10.

Table 10: *Attributes for tagged:operation*

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

tagged:body

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

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

- tagged:field
- tagged: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 11.

Table 11: *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 `taggeded: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 55.

Example 55: *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 56.

Example 56: *Tagged Data Ice Cream Mapping*

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

tagged:sequence

`taggeded:sequence` maps arrays and sequences to a tagged data format. Its three attributes are described in Table 12.

Table 12: *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 57 shows an Artix contract fragment for such a mapping.

Example 57: *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"/>
...
```

Example 57: *Mapping a Sequence to a Tagged Data Format*

```
    <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 13: *Attributes for tagged:choice*

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

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

tagged:case

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

`tagged:case` must contain one child element to describe the mapping of the case's data to a tagged data format. Valid child elements are tagged:field, tagged:sequence, and tagged:choice. Example 58 shows an Artix contract fragment mapping a union to a tagged data format.

Example 58: *Mapping a Union to a Tagged Data Format*

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

Example 58: *Mapping a Union to a Tagged Data Format*

```
    <tagged:choice name="stationPart" descriminatorName="disc">
      <tagged:case name="train">
        <tagged:field name="name"/>
      </tagged:case>
      <tagged:case name="bus">
        <tagged:field name="number"/>
      </tagged:case>
      <tagged:case name="cab">
        <tagged:field name="number"/>
      </tagged:case>
      <tagged:case name="subway">
        <tagged:field name="name"/>
      </tagged:case>
    </tagged:choice>
...
</binding>
...
</definition>
```

Example

Example 59 shows an example of an Artix contract containing a tagged data format binding.

Example 59: *Tagged Data Format Binding*

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

Example 59: *Tagged Data Format Binding (Continued)*

```
      <xsd:simpleType name="widgetSize">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="big"/>
          <xsd:enumeration value="large"/>
          <xsd:enumeration value="mungo"/>
          <xsd:enumeration value="gargantuan"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:complexType name="Address">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street1" type="xsd:string"/>
          <xsd:element name="street2" type="xsd:string"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zipCode" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="widgetOrderInfo">
        <xsd:sequence>
          <xsd:element name="amount" type="xsd:int"/>
          <xsd:element name="order_date" type="xsd:string"/>
          <xsd:element name="type" type="xsd1:widgetSize"/>
          <xsd:element name="shippingAddress" type="xsd1:Address"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="widgetOrderBillInfo">
        <xsd:sequence>
          <xsd:element name="amount" type="xsd:int"/>
          <xsd:element name="order_date" type="xsd:string"/>
          <xsd:element name="type" type="xsd1:widgetSize"/>
          <xsd:element name="amtDue" type="xsd:float"/>
          <xsd:element name="orderNumber" type="xsd:string"/>
          <xsd:element name="shippingAddress" type="xsd1:Address"/>
        </xsd:sequence>
      </xsd:complexType>
    </schema>
  </types>
  <message name="widgetOrder">
    <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
  </message>
  <message name="widgetOrderBill">
    <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
  </message>
```

Example 59: *Tagged Data Format Binding (Continued)*

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

Example 59: *Tagged Data Format Binding (Continued)*

```
        <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>
```

Adding a TibrvMsg Binding

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 section

This section discusses the following topics:

Defining a TibrvMsg Binding	page 204
Defining Array Mapping Policies	page 211
Defining a Custom TibrvMsg Mapping	page 216
Adding Context Information to a TibrvMsg	page 234

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 "TibrvMsg Default Mappings" on page 445.

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 205.
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 216.

TibrvMsg binding namespace

The IONA extensions used to describe TibrvMsg bindings are defined in the namespace `http://schemas.iona.com/transports/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/transports/tibrv"
```

tibrv:binding

`tibrv:binding` is an imediate child of the WSDL `binding` element and identifies that the data is to be packed into a TibrvMsg. Its attributes are described in Table 14.

Table 14: *Attributes for tibrv:binding*

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

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 211.

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 imediate child of a WSDL `operation` element. `tibrv:operation` has no attributes. It can, however, specify an operation specific array policy using a 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 211.

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 imediate 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 15.

Table 15: *Attributes for tibrv:input*

Attribute	Purpose
`messageNameFieldPath`	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 `IT_BUS_MESSAGE_NAME`.
`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. `false` specifies that strings data is passed as `TIRBMSG_STRING`. `true` specifies that string data is passed as `OPAQUE`. This value will override the value set in tibrv:binding.

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 211.

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 216.

tibrv:output

`tibrv:output` is the imediate 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 15.

Table 16: *Attributes for tibrv:output*

Attribute	Purpose
`messageNameFieldPath`	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 `IT_BUS_MESSAGE_NAME`.
`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.

Table 16: *Attributes for tibrv:output*

Attribute	Purpose
stringAsOpaque	An optional attribute that specifies how string data is passed in the message. false specifies that strings data is passed as TIRBMSG_STRING. true specifies that string data is passed as OPAQUE. 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 overides any policies set at the binding level or the operation level. For information on defining array policies see "Defining Array Mapping Policies" on page 211.

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 216.

Example

Example 60 shows an example of an Artix contract containing a default TibrvMsg binding.

Example 60: *Default TibrvMsg 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:tibrv="http://schemas.iona.com/transports/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/">
```

Example 60: *Default TibrvMsg Binding (Continued)*

```
      <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>
  <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 60: *Default TibrvMsg Binding (Continued)*

```
  <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>
```

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 17 shows the effect of setting `tibrv:array` at different levels of a binding.

Table 17: *Effect of tibrv:array*

Child of	Effect
`tibrv:binding`	Sets the array policies for all messages in the binding.
`tibrv:operation`	Array policies set at the operation level only affect the messages defined within the parent `operation` element. They override any array policies set at the binding level.
`tibrv:input`	Array policies set at this level only affect the input message. They override any array policies set at the binding or operation level.
`tibrv:output`	Array policies set at this level only affect the output message. They override any array policies set at the binding or operation level.
`tibrv:msg`	Array policies set at this level affect only the fields defined within the tibrv:msg element. They override any array policies set at higher levels.
`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 18 describes the attributes used to set array policies.

Table 18: *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 `element` element's `name` attribute with a counter. For information on specifying naming expressions see "Custom array naming expressions" on page 213.
`integralAsSingleField`	Specifies how scalar array data is mapped into TibrvMsgField instances. `true`, the default, specifies that arrays are mapped into a single TibrvMsgField. `false` 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. `true` specifies that the number of elements in the array is added as a TibrvMsgField in the same TibrvMsg as the array. `false`, the default, specifies that the number of elements in the array is not included in the TibrvMsg.
`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 `element` element's `name` attribute with `@size`. For information on specifying naming expressions see "Custom array naming expressions" on page 213.

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 are 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 19 shows the available functions for use in building array element names.

Table 19: *Functions Used for Specifying TibrvMsg Array Element Names*

Function	Purpose
`xml:attr('attribute')`	Inserts the value of the named attribute.
`concat(item1, item2, ...)`	Concatenates all of the elements into a single string.

Table 19: *Functions Used for Specifying TibrvMsg Array Element Names*

Function	Purpose
`counter(start, increment)`	Adds an increasing numerical value. The counter starts at *`start`* and increases by *`increment`*.

Example

Example 61 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 61: *TibrvMsg Binding with Array Policies Set*

```
<?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:tibrv="http://schemas.iona.com/transports/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>
```

Example 61: *TibrvMsg Binding with Array Policies Set (Continued)*

```
  <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>
  <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:

- overriding the native XML Schema type specification of contract elements.
- add binding-only elements to the TibrvMsg placed on the wire.
- placing 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 62, 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 62: *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>
```

Example 62: *TibrvMsg Contract Elements (Continued)*

```
<message name="taleRequest">
  <part name="title" type="xsd:sting"/>
</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 the 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 63, 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 63: *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>
  <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 20 shows the matrix of valid casts for native XML Schema types.

Table 20: *Valid Casts for TibrvMsg Binding*

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

Table 20: *Valid Casts for TibrvMsg Binding*

Type	Full Support	Restricted Support[a]
unsignedLong	decimal, string, boolean	byte, unsignedByte, short, usignedShort, int, unsignedInt, long, float, double
float	double, decimal, string, boolean	byte, unsignedByte, short, usignedShort, int, unsignedInt, long, unsignedLong
double	decimal, string, boolean	byte, unsignedByte, short, usignedShort, int, unsignedInt, long, unsignedLong, float
decimal	string, boolean	byte, unsignedByte, short, usignedShort, int, unsignedInt, long, unsignedLong, float, double
string[b]		byte, unsignedByte, short, usignedShort, int, unsignedInt, long, unsignedLong, float, double, decimal, boolean, QName, DateTime
boolean	byte, unsignedByte, short, usignedShort, 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 217, 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 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 64 shows a TibrvMsg binding that adds a static header to each message that is put on the wire.

Example 64: *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 64 would have as its first member a TibrvMsg called header as shown in Example 65.

Example 65: *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;
      }
      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 ending 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 62, would produce the TibrvMsg shown in Example 66.

Example 66: *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;
      }
      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;
      }
    }
```

Example 66: *Default TibrvMsg Example (Continued)*

```
    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 67, you could add it to the binding by redefining the mapping of the entire contract element to include a binding-only element.

Example 67: *TibrvMsg with added 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;
      }
      TibrvMsgField
      {
        name = "pollux";
        id = 0;
        data.str = "This one is a boxxer.";
        size = sizeof(data);
        count = 1;
        type = TIBRVMSG_STRING;
      }
```

Example 67: *TibrvMsg with added TibrvMsg Example*

```
    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;
        }
      }
      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 68 shows a binding for the message shown in Example 67.

Example 68: *TibrvMsg Binding with an Added Binding-only Element*

```
<binging 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"/>
```

Example 68: *TibrvMsg Binding with an Added Binding-only Element*

```
          <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 "TibrvMsg Default Mappings" on page 445.

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 69.

Example 69: *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 70.

Example 70: *TibrvMsg for with a TibrvMsg with a String*

```
TibrvMsg
{
  TibrvMsgField
  {
    name = "grandchild";
    id = 0;
    data.msg =
    {
      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 70 you would use a binding definition similar to the one shown in Example 71.

Example 71: *TibrvMsg Binding with an Extra TibrvMsg Level*

```
<binging 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 62 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 the resembles the one shown in Example 72. The two elements `castor` and `pollux` have been wrapped in a TibrvMsg called `brothers`.

Example 72: *TibrvMsg with Custom TibrvMsg Wrapping*

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

Example 72: *TibrvMsg with Custom TibrvMsg Wrapping (Continued)*

```
      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;
      }
      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 with in 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 73 shows a binding fragment that would create the TibrvMsg shown in Example 72.

Example 73: *Binding of a Complex Type with an Extra TibrvMsg Level*

```
<binging 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 21.

Table 21: *Attributes for tibrv:msg*

Attribute	Purpose
name	Specifies the name of the contract element which this TibrvMsg instance gets its value. If this attribute is not present, then the TibrvMsg is considered a binding-only element.
alias	Specifies the value of the name member of the TibrvMsg instance. If this attribute is not specified, then the binding will use the value of the name attribute.

Table 21: *Attributes for tibrv:msg (Continued)*

Attribute	Purpose
`element`	Used only when `tibrv:msg` is an immediate child of `tibrv:context`. Specifies the QName of the element defining the context data to use when populating the TibrvMsg. See “Adding Context Information to a TibrvMsg” on page 234.
`id`	Specifies the value of the `id` member of the TibrvMsg instance. The default value is `0`.
`minOccurs/` `maxOccurs`	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 22.

Table 22: *Attributes for tibrv:field*

Attribute	Purpose
`name`	Specifies the name of the contract element which this TibrvMsgField instance gets its value. If this attribute is not present, then the TibrvMsgField is considered a binding-only element.
`alias`	Specifies the value of the `name` member of the TibrvMsgField instance. If this attribute is not specified, then the binding will use the value of the `name` attribute.
`element`	Used only when `tibrv:field` is an immediate child of `tibrv:context`. Specifies the QName of the element defining the context data to use when populating the TibrvMsgField. See “Adding Context Information to a TibrvMsg” on page 234.
`id`	Specifies the value of the `id` member of the TibrvMsgField instance. The default value is `0`.

Table 22: *Attributes for tibrv:field*

Attribute	Purpose
`type`	Specifies the XML Schema type of the data being used to populate the `data` member of the TibrvMsgField instance. For a list of supported types, see “TibrvMsg Default Mappings” on page 445.
`value`	Specifies the value inserted into the `data` member of the TibrvMsgField 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 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 are some application specific information you need to abide by:

- At least one piece of the integrated solution must me 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 74 shows a contract fragment containing the appropriate entries for this scenario.

Example 74: *Using Context Data in a TibrvMsg Binding*

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

Example 74: *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="correalatedService">
  ...
  </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 responses' `corrID` from the response context. If the Artix side of your application is a server, you will need to include logic to read requests' `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.

Adding a Pure XML Binding

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. See "Hand editing" on page 239.

Using Artix Designer

To add an XML binding from the Artix Designer Diagram view:

1. Right-click the **Bindings** node to activate the pop-up window.
2. Select **New Binding**to bring up the **Select Source Resources** window.
3. Select at least one contract from the list to act as a source for interfaces.

   All of the interfaces in the selected contracts will be made available to you later. The contracts will also be imported to the target resource using WSDL `import` elements.
4. Click **Next** to bring up the **Select Binding Type** window.
5. Select **XML**.

6. Click **Next** to bring up the **Set Binding Defaults** window, shown in Figure 64.

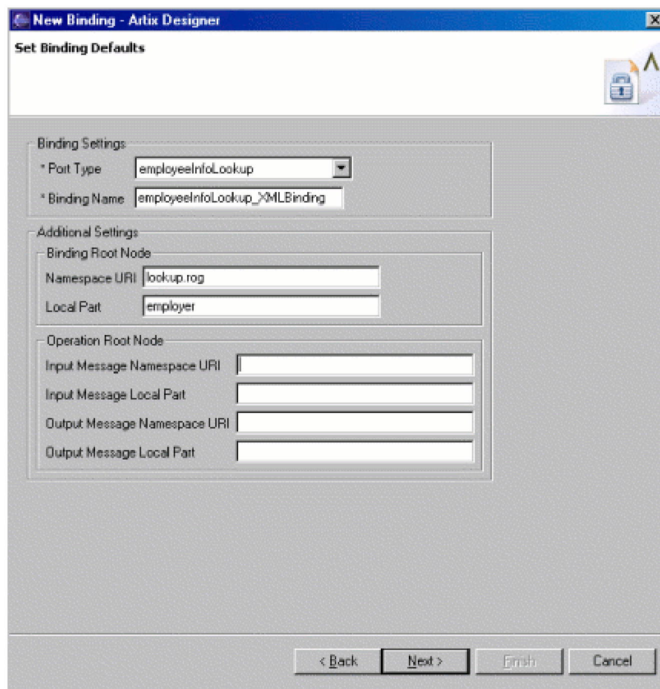


Figure 64: *Setting the Default Values for an XML Binding*

7. Select the interface to mapped to the binding from the **Port Type** drop-down list.
8. Enter a name for the binding in the **Binding Name** field.
9. Enter an optional QName for the root node of the generated XML data by filling in both the **Namespace URI** field and the **Local Part** field.
10. Under **Operation Root Node**, enter optional QNames for the root nodes of the input and output messages for the operations defined by the interface.
11. Click **Finish**.

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 239.
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 240.
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://schemas.iona.com/bindings/xmlformat`. Artix tools use the prefix `xformat` to represent the XML binding extensions. Add the following line to your contracts:

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

XML messages on the wire

When you specify that an interface's message 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, without the `rootNode` attribute set the message defined in Example 75 would generate an XML document with the root element `lineNumer`.

Example 75: *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 without the `rootNode` attribute set. However, the message in Example 76 would generate an invalid XML document.

Example 76: *Invalid XML Binding Message*

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
  <element name="entryNum" type="xsd:int"/>
  ...
</types>
```

Example 76: *Invalid XML Binding Message (Continued)*

```
<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 77 for the message defined in Example 76. The Artix generated XML document is invalid because it has two root elements: `pairName` and `entryNum`.

Example 77: *Invalid XML Document*

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

If you set the `rootNode` attribute, as shown in Example 78 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 78: *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>
```

A an XML document generated from the input message would be similar to Example 79. Notice that the XML document now only has one root element.

Example 79: *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 78 to have a different root element from the input message, you could override the binding's root element as shown in Example 80.

Example 80: *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>
```

Adding a G2++ Binding

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 may 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 81: *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 82: *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/">
```

Example 82: *WSDL Logical Description of ERecord Message (Continued)*

```
      <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-structure 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 83: *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 with 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 84: *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 85: *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>
```

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 86: *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 87: *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.

Adding Transports

To fully define a service you need to describe at least one transport.

In this chapter

This chapter discusses the following topics:

Introducing Services	page 252
Defining a Service	page 253
Creating an HTTP Service	page 255
Creating a CORBA Service	page 264
Creating an IIOP Service	page 273
Creating a WebSphere MQ Service	page 279
Creating a Java Messaging System Service	page 288
Adding a TIBCO Service	page 298
Creating a Tuxedo Service	page 305
Configuring a Service to Use Codeset Conversion	page 309

Introducing Services

Overview

To complete the definition of a service, you need to describe the transport details used to connect the service to a network. Transport details are defined inside a `port` element. Each port specifies the address and configuration information for connecting the application to a network.

Ports are grouped within `service` elements. A service can contain one or more ports. The convention is that the ports defined within a particular service are related in some way. For example all of the ports might be bound to the same port type, but use different network protocols, like HTTP and WebSphere MQ.

Defining a Service

Overview

All of the transport details for an endpoint are defined in `service` elements. A `service` element defines a collection of `port` elements. The `port` elements defines the relationship between a particular `binding` element and the transport on which the messages are to be sent. The `port` element contains all of the information defining the endpoints connection to a network including what type of transport to use, the address, and any other transport details.

The service element

The `service` element contains a group of one or more ports that have some relationship. How the ports are related is up to you. For example you could build a contract where every port is contained in its own `service`, or you could decide to group all of the ports that are bound to a particular interface into `service` elements.

A `service` element has one required attribute, `name`, that identifies the service. The identifier must be unique among all of the services defined in the contract. Example 88 shows an example of a service named `riotService`.

Example 88: *Sample Service*

```
<service name="riotService">
  <port ....>
    ...
  </port>
</service>
```

The port element

The `port` element defines how a binding is tied to a specific network transport. You specify the binding from which messages will be sent over the network using the `port` element's `binding` attribute. The value of the binding attribute must correspond to a binding defined with in the same contract, or a contract imported into the same contract, in which the port is defined.

The port element also has an attribute, `name`, that identifies the port. The identifier must be unique among the ports describe within the containing `service` element. shows a `port` element, `riotPort`, that defines a port bound to `riotBinding`.

Example 89: *Sample Port*

```
<service name="riotService">
  <port name="riotPort" binding="riotBinding">
    ...
  </port>
</service>
```

Contained within the `port` element are the elements used to define the details of the transport that is used to send messages. In a standard WSDL contract the transport details would be represented using a `soap:address` element. However, Artix provides a number of transports and the elements to define them. The following sections describe the details of adding the details for these transports.

Creating an HTTP Service

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.

Artix provides two ways of specifying an HTTP service'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 in the contract.

soap:address

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

Example 90 shows a port used to send SOAP over HTTP.

Example 90: *SOAP Port*

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

http:address

When your messages are formatted using any other payload format than SOAP, such as fixed, you must use Artix's `http:address` element to specify the service's address.

Using Artix Designer

To add an HTTP port to your contract from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.

4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window, shown in Figure 65.

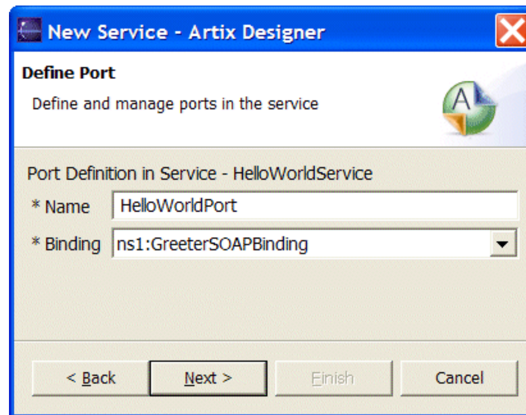


Figure 65: *Specifying the Name and Binding for a Port*

7. Enter a name for the new port in the **Name** field.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next** to bring up the **Define Port Properties** window.
10. Select **http** from the **Transport Type** drop-down list.

Note: If the payload being transmitted is SOAP, then select **soap**.

11. The http transport attributes editor will appear as shown in Figure 66.

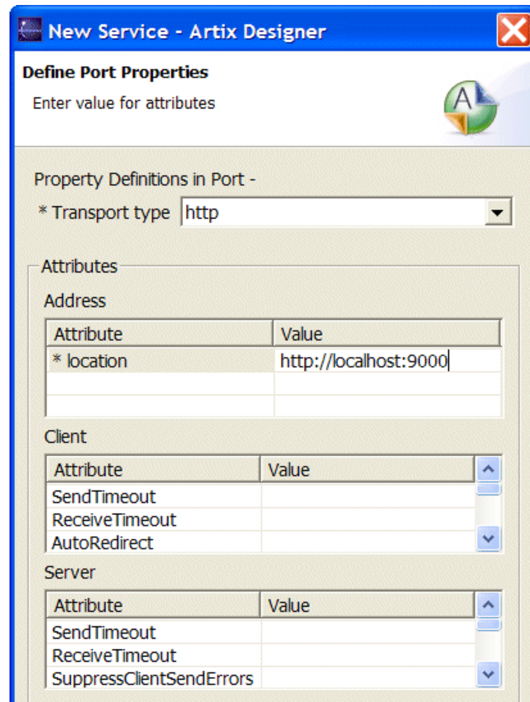


Figure 66: *Setting the HTTP Transport Attributes*

12. Enter the HTTP address for the port in the **location** field of the **Address** table.
13. Enter values for any of the optional configuration settings you desire. See "HTTP Port Properties" on page 451.
14. Click **Finish**.

Using the command line tool

You can use the wsdltoservice command line tool to add an HTTP service definition to an Artix contract. To use wsdltoservice to add an HTTP service use the tool with the following options.

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

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

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

`-hshka` *`honorKeepAlive`*	Specifies if the server honors client keep-alive requests. Valid values are `true` or `false`.
`-hsmps` *`serverMultiplexPoolSize`*	
`-hsrurl` *`redirectURL`*	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource.
`-hscl` *`contentLocation`*	Specifies the URL where the resource being sent in a server response is located.
`-hsce` *`contentEncoding`*	Specifies what additional content codings have been applied to the information being sent by the server, and what decoding mechanisms the client therefore needs to retrieve the information.
`-hsst` *`serverType`*	Specifies what type of server is sending the response to the client.
`-hssc` *`serverCentificate`*	Specifies the full path to the X509 certificate issued by the certificate authority for the server.
`-hsscc` *`serverCentificateChain`*	Specifies the full path to the file that contains all the certificates in the chain.
`-hsspk` *`serverPrivateKey`*	Specifies the full path to the private key that corresponds to the X509 certificate specified by *`serverCertificate`*.
`-hsspkp` *`serverPrivateKeyPassword`*	Specifies a password that is used to decrypt the private key.
`-hcst` *`clientSendTimeout`*	Specifies the number of milliseconds that the client can continue to try to send a request to the server before the connection is timed out.
`-hccvt` *`clientReceiveTimeout`*	Specifies the number of milliseconds that the client can continue to try to receive a response from the server before the connection is timed out.
`-hctrc` *`trustedRootCertificates`*	Specifies the full path to the X509 certificate for the certificate authority.
`-hcuss` *`ueSecureSockets`*	Specifies if the client uses secure sockets. Valid values are *`true`* or *`false`*.

`-hcct` *`contentType`*	Specifies the media type of the data being sent in the body of the client request.
`-hccc` *`clientCacheControl`*	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
`-hcar` *`autoRedirect`*	Specifies if the server should automatically redirect client requests.
`-hcun` *`userName`*	Specifies the username the client uses to register with servers.
`-hcp` *`password`*	Specifies the password the client uses to register with servers.
`-hcat` *`clientAuthorizationType`*	Specifies the authorization mechanisms the client uses when contacting servers.
`-hca` *`clientAuthorization`*	Specifies the authorization credentials used to perform the authorization.
`-hca` *`accept`*	Specifies what media types the client is prepared to handle.
`-hcal` *`acceptLanguage`*	Specifies what language the client prefers for the purposes of receiving a response
`-hcae` *`acceptEncoding`*	Specifies what content codings the client is prepared to handle.
`-hch` *`host`*	Specifies the internet host and port number of the resource on which the client request is being invoked.
`-hccn` *`clientConnection`*	Specifies if the client will open a new connection for each request or if it will keep the original one open. Valid values are `close` and `Keep-Alive`.
`-hcck` *`cookie`*	Specifies a static cookie to be sent to the server.
`-hcbt` *`browserType`*	Specifies information about the browser from which the client request originates.
`-hcr` *`referer`*	Specifies the value for the client's referring entity.
`-hcps` *`proxyServer`*	Specifies the URL of the proxy server, if one exists along the message path.

`-hcpun` *`proxyUserName`*	Specifies the username that the client uses to authorize with proxy servers.
`-hcpp` *`proxyPassword`*	Specifies the password that the client uses to authorize with proxy servers.
`-hcpat` *`proxyAuthorizationType`*	Specifies the authorization mechanism the client uses with proxy servers.
`-hcpa` *`proxyAuthorization`*	Specifies the actual data that the proxy server should use to authenticate the client.
`-hccce` *`clientCertificate`*	Specifies the full path to the X509 certificate issued by the certificate authority for the client.
`-hcccc` *`clientCertificateChain`*	Specifies the full path to the file that contains all the certificates in the chain.
`-hcpk` *`clientPrivateKey`*	Specifies the full path to the private key that corresponds to the X509 certificate specified by *`clientCertificate`*.
`-hcpkp` *`clientPrivateKeyPassword`*	Specifies a password that is used to decrypt the private key.
`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d dir`	Specifies the output directory for the generated contract.
`-L file`	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Example

Example 91 shows the namespace entries you need to add to the `definitions` element of your contract to use the HTTP extensions.

Example 91: *Artix HTTP Extension Namespaces*

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

The `http:address` element is similar to the `soap:address` element. It has one attribute, `location`, that specifies the service's address as a URL. Example 92 shows a port used to send fixed data over HTTP.

Example 92: *Generic HTTP Port*

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

Creating a CORBA Service

Overview

Generally, when you are creating a CORBA service with Artix, you need to do two things. First, you must configure the Artix 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 Artix service.

In this section

This section discusses the following topics:

Configuring an Artix CORBA Port	page 265
Generating CORBA IDL	page 271

Configuring an Artix CORBA Port

Overview

CORBA ports 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 port 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 objects IOR directly, by entering the object's IOR directly into 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 Deploying and Managing Artix Solutions.

- 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 POA policies

Using the optional `corba:policy` element, you can describe a number of POA polices the Artix service 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 POA's 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 Artix Designer

To add a CORBA service to your contract from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.
4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window, shown in Figure 65 on page 256
7. Enter a name for the new port in the **Name** field.
8. Select a CORBA binding to be exposed by this port from the **Binding** drop-down list.

> **Note:** Because CORBA defines both a payload format and a transport, you can only expose a CORBA binding through a CORBA endpoint. If you do not choose a CORBA binding, Artix Designer will not allow you to select CORBA as your transport.

9. Click **Next** to bring up the **Define Port Properties** window, shown in Figure 67.

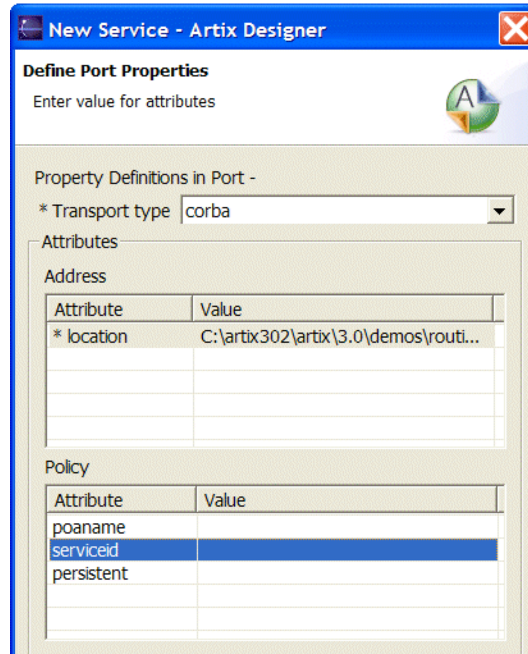


Figure 67: *Specifying the Address of a CORBA Endpoint*

10. Enter a valid CPRBA address in the **location** field of the **Address** table. See “CORBA address specification” on page 265.
11. Set any desired POA policies in the **Policy** table. See “Specifying POA policies” on page 266.
12. Click **Finish**.

Using the command line tool

You can use the `wsdltoservice` command line tool to add a CORBA service definition to an Artix contract. To use `wsdltoservice` to add a CORBA service 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]  wsdlurl
```

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

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

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to Example 94:

Example 93: *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 exist 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 binding and port type. 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:

`-idl`	Instructs the tool to generate an IDL file from the specified binding.
`-b` *`binding`*	Specifies the CORBA binding from which to generate IDL.
`-corba`	Instructs the tool to generate a CORBA binding for the specified port type.
`-i` *`portType`*	Specifies the name of the port type being mapped to a CORBA binding.
`-d` *`dir`*	Specifies the directory into which the new WSDL file is written.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl_file`*`.idl`.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	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.

Creating an IIOP Service

Overview

Artix allows you to use IIOP as a generic transport for send data using any of the payload formats. When using IIOP as a generic transport, you define your service'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.

> **Note:** Generic IIOP is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports IIOP.

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.

Using Artix Designer

To add an IIOP port from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.
4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window.
7. Enter a name for the new port in the **Name** field.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next** to bring up the **Define Port Properties** window.

10. Select **tunnel** from the **Transport Type** drop-down list to bring up the IIOP tunnel transport properties editor shown in Figure 68.

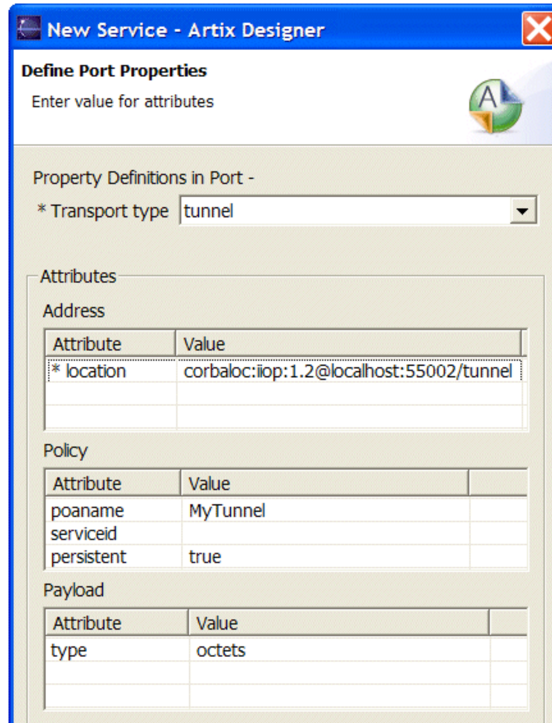


Figure 68: *Editing IIOP Tunnel Transport Attributes*

11. Enter a valid IIOP address in the **location** field of the **Address** table. See “IIOP address specification” on page 275.
12. Set any desired POA policies in the **Policy** table. See “Specifying POA policies” on page 275.
13. Specify the type of payload being sent through the tunnel in the **type** field of the **Payload** table. See “Specifying type of payload encoding” on page 275.
14. Click **Finish**.

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 objects IOR directly, by entering the object's IOR directly into 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 Deploying and Managing Artix Solutions.

- 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 turned off so that the agents sending the message maintain complete control over codeset conversion. If you wish to turn automatic codeset negotiation on use the following:

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

Specifying POA policies

Using the optional `iiop:policy` element, you can describe the POA polices the Artix service will use when creating the IIOP port. 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 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 of the POA that Artix creates for the IIOP port, 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 POA's 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 port'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

You can use the `wsdltoservice` command line tool to add an IIOP service definition to an Artix contract. To use `wsdltoservice` to add an IIOP service 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] [-q] [-h] [-V]  wsdlurl
```

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

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

Example

For example, an IIOP port for the `personalInfoLookup` binding would look similar to Example 94:

Example 94: *CORBA personalInfoLookup Port*

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

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

Creating a WebSphere MQ Service

Overview

The description for an Artix WebSphere MQ port 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 ports and their attributes:

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

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

WebSphere MQ namespace

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

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

Required attributes

When you define a WebSphere MQ service you need to provide at least enough information for the service to connect to its message queues. For any WebSphere application that means setting the `QueueManager` and `QueueName` attributes of the port. 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 it.

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 Artix application is going to use remote queues. This setting instructs Artix to load `libmqic` instead of `libmqm`.

Using Artix Designer

To add a WebSphere MQ port from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.
4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window.
7. Enter a name for the new port in the **Name** field.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next** to bring up the **Define Port Properties** window.

10. Select **mq** from the **Transport Type** drop-down list to bring up the WebSphere MQ transport properties editor shown in Figure 69.

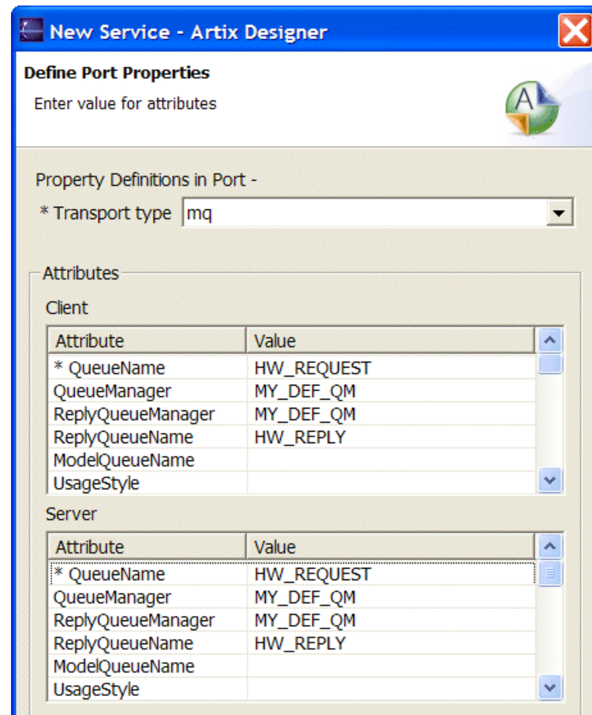


Figure 69: *Editing WebSphere MQ Transport Attributes*

11. If you are adding a port for an MQ client, enter the name of the client's request queue in the **QueueName** field of the **Client** table.
12. If you are adding a port for an MQ client, enter the name of the queue manager for the client's request queue in the **QueueManager** field of the **Client** table.
13. If you are adding a port for an MQ client that will be getting responses from its server, enter the name of the client's reply queue in the **ReplyQueueName** field of the **Client** table.

14. If you are adding a port for an MQ client that will be getting responses from its server, enter the name of the queue manager for the client's reply queue in the **ReplyQueueManager** field of the **Client** table.
15. If you are adding a port for an MQ server, enter the name of the server's response queue in the **QueueName** field of the **Server** table.
16. If you are adding a port for an MQ server, enter the name of the queue manager for the server's response queue in the **QueueManager** field of the **Server** table.
17. Edit any of the remaining optional attributes. See "WebSphere MQ Port Properties" on page 493
18. Click **Finish**.

Using the command line tool

You can use the `wsdltoservice` command line tool to add a WebSphere MQ service definition to an Artix contract. To use `wsdltoservice` to add a WebSphere MQ service use the tool with the following options.

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

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

`-e` *`service`*	Specifies the name of the generated service.
`-t` *`port`*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *`binding`*	Specifies the name of the binding for which the service is generated.
`-sqm` *`queueManager`*	Specifies the name of the server's queue manager.
`-sqn` *`queue`*	Specifies the name of the server's request queue.
`-srqm` *`queueManager`*	Specifies the name of the server's reply queue manager.
`-srqn` *`queue`*	Specifies the name of the server's reply queue.
`-smqn` *`modelQueue`*	Specifies the name of the server's model queue.
`-sus` *`usageStyle`*	Specifies the value of the server's `UsageStyle` attribute. Valid values are `Peer`, `Requester`, or `Responder`.
`-scs` *`correlationStyle`*	Specifies the value of the server's `CorrelationStyle` attribute. Valid values are `messageId`, `correlationId`, or `messageId copy`.
`-sam` *`accessMode`*	Specifies the value of the server's `AccessMode` attribute. Valid values are `peek`, `send`, `receive`, `receive exclusive`, or `receive shared`.
`-sto` *`timeout`*	Specifies the value of the server's `Timeout` attribute.
`-sme` *`expiry`*	Specifies the value of the server's `MessageExpiry` attribute.
`-smp` *`priority`*	Specifies the value of the server's `MessagePriority` attribute.
`-smi` *`messageId`*	Specifies the value of the server's `MessageId` attribute.
`-sci` *`correlationId`*	Specifies the value of the server's `CorrelationId` attribute.
`-sd` *`delivery`*	Specifies the value of the server's `Delivery` attribute.

`-st` *`transactional`*	Specifies the value of the server's `Transactional` attribute. Valid values are `none`, `internal`, or `xa`.
`-sro` *`reportOption`*	Specifies the value of the server's `ReportOption` attribute. Valid values are `none`, `coa`, `cod`, `exception`, `expiration`, or `discard`.
`-sf` *`format`*	Specifies the value of the server's `Format` attribute.
`-sad` *`applicationData`*	Specifies the value of the server's `ApplicationData` attribute.
`-sat` *`accountingToken`*	Specifies the value of the server's `AccountingToken` attribute.
`-scn` *`connectionName`*	Specifies the name of the connection by which the adapter connects to the queue.
`-sc` *`convert`*	Specifies if the messages in the queue need to be converted to the system's native encoding. Valid values are `true` or `false`.
`-scr` *`reusable`*	Specifies the value of the server's `ConnectionReusable` attribute. Valid values are `true` or `false`.
`-scfp` *`fastPath`*	Specifies the value of the server's `ConnectionFastPath` attribute. Valid values are `true` or `false`.
`-said` *`idData`*	Specifies the value of the server's `ApplicationIdData` attribute.
`-saod` *`originData`*	Specifies the value of the server's `ApplicationOriginData` attribute.
`-cqm` *`queueManager`*	Specifies the name of the client's queue manager.
`-cqn` *`queue`*	Specifies the name of the client's request queue.
`-crqm` *`queueManager`*	Specifies the name of the client's reply queue manager.
`-crqn` *`queue`*	Specifies the name of the client's reply queue.
`-cmqn` *`modelQueue`*	Specifies the name of the client's model queue.
`-cus` *`usageStyle`*	Specifies the value of the client's `UsageStyle` attribute. Valid values are `Peer`, `Requester`, or `Responder`.

`-ccs` *correlationStyle*	Specifies the value of the client's `CorrelationStyle` attribute. Valid values are `messageId`, `correlationId`, or `messageId copy`.
`-cam` *accessMode*	Specifies the value of the client's `AccessMode` attribute. Valid values are `peek`, `send`, `receive`, `receive exclusive`, or `receive shared`.
`-cto` *timeout*	Specifies the value of the client's `Timeout` attribute.
`-cme` *expiry*	Specifies the value of the client's `MessageExpiry` attribute.
`-cmp` *priority*	Specifies the value of the client's `MessagePriority` attribute.
`-cmi` *messageId*	Specifies the value of the client's `MessageId` attribute.
`-cci` *correlationId*	Specifies the value of the client's `CorrelationId` attribute.
`-cd` *delivery*	Specifies the value of the client's `Delivery` attribute.
`-ct` *transactional*	Specifies the value of the client's `Transactional` attribute. Valid values are `none`, `internal`, or `xa`.
`-cro` *reportOption*	Specifies the value of the client's `ReportOption` attribute. Valid values are `none`, `coa`, `cod`, `exception`, `expiration`, or `discard`.
`-cf` *format*	Specifies the value of the client's `Format` attribute.
`-cad` *applicationData*	Specifies the value of the client's `ApplicationData` attribute.
`-cat` *accountingToken*	Specifies the value of the client's `AccountingToken` attribute.
`-ccn` *connectionName*	Specifies the name of the connection by which the adapter connects to the queue.
`-cc` *convert*	Specifies if the messages in the queue need to be converted to the system's native encoding. Valid values are `true` or `false`.
`-ccr` *reusable*	Specifies the value of the client's `ConnectionReusable` attribute. Valid values are `true` or `false`.

`-ccfp` *`fastPath`*	Specifies the value of the client's `ConnectionFastPath` attribute. Valid values are `true` or `false`.
`-caid` *`idData`*	Specifies the value of the client's `ApplicationIdData` attribute.
`-caod` *`originData`*	Specifies the value of the client's `ApplicationOriginData` attribute.
`-caqn` *`queue`*	Specifies the remote queue to which a server will put replies if its queue manager is not on the same host as the client's local queue manager.
`-cui` *`userId`*	Specifies the value of the client's `UserIdentification` attribute.
`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *`dir`*	Specifies the output directory for the generated contract.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Example

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 95.

Example 95: *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>
```

More information

For a detailed description of the WebSphere MQ transport configuration attributes see “WebSphere MQ Port Properties” on page 493.

Creating a Java Messaging System Service

Overview

Artix provides a transport plug-in that enables systems to place and receive messages from 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 JNDI to locate and obtain references to the JMS provider that brokers for the JMS destination with which it wants to connect. 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 the payload formatting and packages it into either a JMS `ObjectMessage` or a `TextMessage`. When a message is packaged as an `ObjectMessage` the message information, 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 information, 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 port is SOAP, and the messages are packaged as `TextMessage`, the receiving JMS application will get a text message containing all of the SOAP envelope information. For a message encoded using the fixed binding, the message will contain no formatting information, simply a string of characters, numbers, and spaces.

Port configuration

The JMS port configuration is done by using a `jms:address` element in your service's `port` description. `jms:address` uses the attributes described in Table 23 to configure the JMS connection.

Table 23: *Required JMS Port Attributes*

Attribute	Description
`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.
`jndiProviderURL`	Specifies the URL of the JNDI service where the connection information for the JMS destination is stored.
`initialContextFactory`	Specifies the name of the `InitialContextFactory` class or a list of package prefixes used to construct URL context factory classnames. For more details on specifying a JNDI `InitialContextFactory`, see "JNDI InitialContextFactory settings" on page 296.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which requests are sent.
`jndiReplyDestinationName`	Specifies the JNDI name bound to 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".
`connectionUserName`	Specifies the username to use when connecting to a JMS broker.
`connectionPassword`	Specifies the password to use when connecting to a JMS broker.

Table 23: *Required JMS Port Attributes*

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

Using a named reply destination

By default Artix endpoints using JMS create a temporary queue for sending replies back and forth. 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.

Using correlation IDs

If you want to configure Artix to use JMS message IDs as the correlation IDs you can set the optional useMessageIDAsCorrelationID attribute to true. The default for this attribute is false.

Setting up durable subscriptions

If you want to configure your Artix service 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 Artix service 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 Artix service 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 service crashes while processing a request or the 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 the message, pull it from the queue again, and attempt to resend it.

Optional JNDI settings

To increase interoperability with JMS and JNDI providers, the `<jms:address>` element has a number of optional attributes to facilitate configuring a JNDI connection. These optional attributes are:

- `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 using these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Using Artix Designer

To add a JMS port from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.
4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window.
7. Enter a name for the new port in the **Name** field.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next** to bring up the **Define Port Properties** window.

10. Select **jms** from the **Transport Type** drop-down list to bring up the JMS transport properties editor shown in Figure 70.

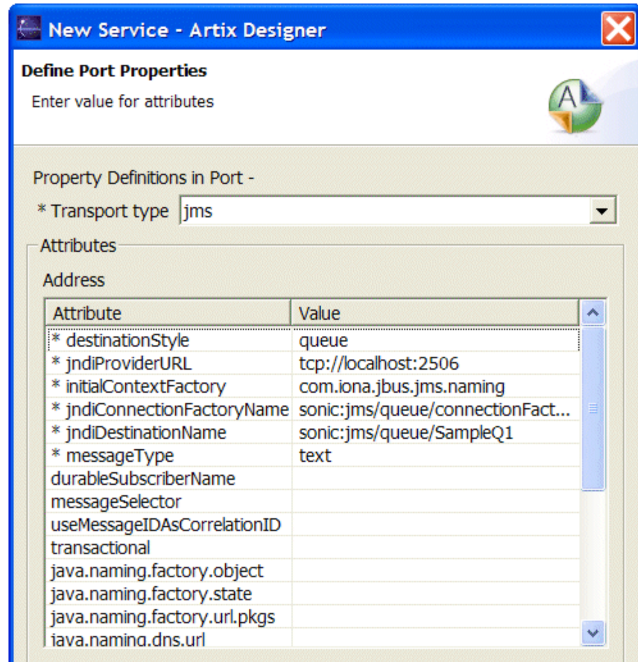


Figure 70: *Editing JMS Transport Attributes*

11. Set the required port properties. See Table 23 on page 289.
12. Click **Finish**.

Using the command line tools

You can use the `wsdltoservice` command line tool to add a JMS service definition to an Artix contract. To use `wsdltoservice` to add a JMS service use the tool with the following options.

```
wsdltoservice -transport jms [-e service] [-t port] [-b binding]
              [-jds destionationStyle] [-jpu jndiProviderURL]
              [-jcf initialContextFactory]
              [-jfn jndiConnectionFactoryName]
              [-jdn jndiDestinationName] [-jmt messageType]
              [-o file] [-d dir] [-L file] [-q] [-h] [-V]  wsdlurl
```

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

`-e` *service*	Specifies the name of the generated service.
`-t` *port*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *binding*	Specifies the name of the binding for which the service is generated.
`-jds` *destinationStyle*	Specifies if the JMS destination is a JMS queue or a JMS topic.
`-jpu` *providerURL*	Specifies the URL of the JNDI service where the connection information for the JMS destination is stored.
`-jcf` *initialContextFactory*	Specifies the name of the `InitialContextFactory` class or a list of package prefixes used to construct URL context factory classnames. For more details on specifying a JNDI `InitialContextFactory`, see "JNDI InitialContextFactory settings" on page 296.
`-jfn` *connectionFactoryName*	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`-jdn` *destinationName*	Specifies the JNDI name bound to the JMS destination to which Artix connects.

-jmt *messageType*	Specifies how the message data will be packaged as a JMS message. Valid values are `text` or `binary`.
-o *file*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
-d *dir*	Specifies the output directory for the generated contract.
-L *file*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
-q	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
-h	Specifies that the tool will display a usage message.
-V	Specifies that the tool runs in verbose mode.

Example

Example 96 shows an example of an Artix JMS port specification.

Example 96: *Artix JMS Port*

```
<service name="HelloWorldService">
  <port binding="tns:HelloWorldPortBinding" name="HelloWorldPort">
    <jms:address destinationStyle="queue"
                 jndiProviderURL="tcp://localhost:2506"
                 initialContextFactory="com.sonicsw.jndi.mfcontext.MFContextFactory"
                 jndiConnectionFactoryName="QCF"
                 jndiDestinationName="testQueue"
                 messageType="text"/>
  </port>
</service>
```

JNDI InitialContextFactory settings

The usual method of specifying the JNDI is to enter the class name provided by your JNDI provider. In Example 96, the JMS port is using the JNDI provided with SonicMQ and the class specified, `com.sonicsw.jndi.mfcontext.MFContextFactory`, is the class used by Sonic's JNDI server to create a JNDI context.

Alternatively, you can specify a colon-separated list of package prefixes to use when loading URL context factories. The JNDI service takes each package prefix and appends the URL schema name to form a sub-package. It then prepends the URL schema name to `URLContextFactory` to form a class name within the sub-package. Once the new class name is formed, the JNDI service then tries to instantiate the class using the newly formed name. For example, if your Artix contract described the JMS port shown in Example 97, the JNDI service would instantiate a context factory with the class name `com.iona.jbus.jms.naming.sonic.sonicURLContextFactory` to perform lookups.

Example 97: *JMS Port with Alternate InitialContextFactory Specification*

```
<service name="HelloWorldService">
  <port binding="tns:HelloWorldPortBinding" name="HelloWorldPort">
    <jms:address destinationStyle="queue"
                 jndiProviderURL="tcp://localhost:2506"
                 initialContextFactory="com.iona.jbus.jms.naming"
                 jndiConnectionFactoryName="sonic:jms/queue/connectionFactory"
                 jndiDestinationName="sonic:jms/queue/helloWorldQueue"
                 messageType="text"/>
  </port>
</service>
```

The `URLContextFactory` then uses the URL specified in the `jndiConnectionFactoryName` and the `jndiDestinationFactoryName` attributes to obtain references to the desired JMS `ConnectionFactory` and the desired JMS `Destination`. The JNDI service is completely bypassed using this method and allows you to connect to JMS implementations that do not use JNDI or to connect to JMS `Destination` that are not registered with the JNDI service.

So instead of looking up the JMS `ConnectionFactory` using the JNDI name bound to it, Artix will get a reference directly to `ConnectionFactory` using the name given to it when it was created. Using the contract in Example 97,

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.

Note: Due to a known bug in the SonicMQ JNDI service, it is recommended that you use this method of specifying the `InitialContextFactory` when using SonicMQ.

Adding a TIBCO Service

Overview

The TIBCO Rendezvous transport lets you use Artix to integrate systems based on TIBCO Rendezvous (TIB/RV) software.

Supported Features

Table 24 shows the matrix of TIBCO Rendezvous features Artix supports.

Table 24: *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 describe the port using TIB/RV in the physical part of an Artix contract. The extensions used to describe a TIB/RV port are defined in the namespace:

```
xmlns:tibrv="http://schemas.iona.com/transports/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 description is contained within a `port` element. Artix uses `tibrv:port` to describe the attributes of a TIB/RV port. The only required attribute for a `tibrv:port` is `serverSubject` which specifies the subject to which the server listens.

Using Artix Designer

To add a TIB/RV port from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.
4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window.
7. Enter a name for the new port in the **Name** field.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next** to bring up the **Define Port Properties** window.

10. Select **tibrv** from the **Transport Type** drop-down list to bring up the TIBCO transport properties editor shown in Figure 71.

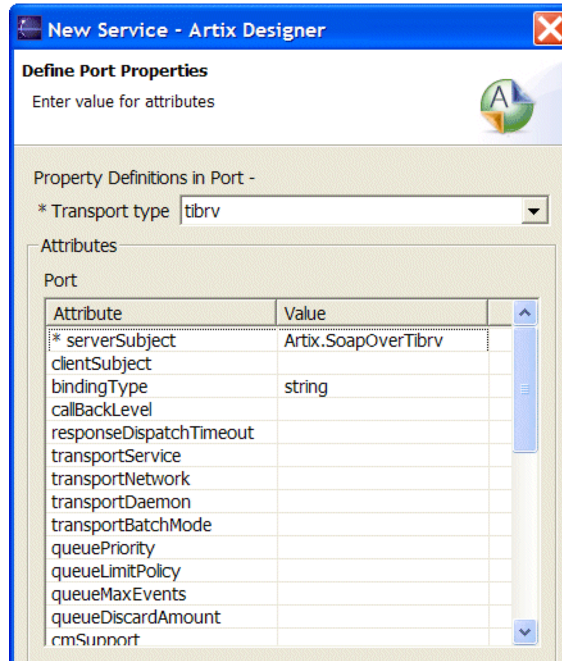


Figure 71: *Editing TIBCO Transport Attributes*

11. Specify the name of the subject to which the server listens in the **serverSubject** field.
12. Set any desired optional attributes. See "Tibco Port Properties" on page 533.
13. Click **Finish**.

Using the command line tools

You can use the wsdltoservice command line tool to add a TIB/RV service definition to an Artix contract. To use wsdltoservice to add a TIB/RV service use the tool with 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]
              [-tctln cmTransportLedgerName]
              [-tctsl cmTransportSyncLedger]
              [-tctra cmTransportRelayAgent]
              [-tctdtl cmTransportDefaultTimeLimit]
              [-tclca cmListenerCancelAgreements]
              [-tcqtsn cmQueueTransportServerName]
              [-tcqtcn cmQueueTransportClientName]
              [-tcqtww cmQueueTransportWorkerWeight]
              [-tcqtws cmQueueTransportWorkerTasks]
              [-tcqtsw cmQueueTransportSchedulerWeight]
              [-tcqtsh cmQueueTransportSchedulerHeartbeat]
              [-tcqtsa cmQueueTransportSchedulerActivation]
              [-tcqtct cmQueueTransportCompleteTime]
              [-tmnfv messageNameFieldValue]
              [-tmnfp messageNameFieldPath]
              [-tbfi bindingFieldId] [-tbfn bindingFieldName]
              [-o file] [-d dir] [-L file]
              [-q] [-h] [-V] wsdlurl
```

The -transport tibrv flag specifies that the tool is to generate a TIB/RV service. The other options are as follows.

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

`-tss` *subject*	Specifies the subject to which the server listens.
`-tbt` *bindingType*	Specifies the message binding type. Valid vales are `msg`, `xml`, `opaque`, or `string`.
`-tcl` *callbackLevel*	Specifies the server-side callback level when TIB/RV system advisory messages are received. Valid values are `INFO`, `WARN`, or `ERROR`.
`-trdt` *timeout*	Specifies the client-side response receive dispatch timeout.
`-tts` *transportService*	Specifies the UDP service name or port for TibrvNetTransport.
`-ttn` *transportNetwork*	Specifies the binding network addresses for TibrvNetTransport.
`-ttbm` *batchMode*	Specifies if the TIB/RV transport uses batch mode to send messages. Valid values are `DEFAULT_BATCH` and `TIMER_BATCH`.
`-tqp` *priority*	
`-tqlp` *queueLimitPolicy*	Valid values are `DISCARD_NONE`, `DISCARD_NEW`, `DISCARD_FIRST`, or `DISCARD_LAST`.
`-tqme` *queueMaxEvents*	
`-tqda` *queueDiscardAmount*	
`-tcs` *cmSupport*	Specifies if Certified Message Delivery support is enabled. Valid values are `true` or `false`.
`-tctsn` *cmTransportServerName*	Specifies the server's TibrvCmTransport correspondent name.
`-tctcn` *cmTransportClientName*	Specifies the client TibrvCmTransport correspondent name.
`-tctro` *cmTransportRequestOld*	Specifies if the endpoint can request old messages on start-up. Valid values are `true` or `false`.
`-tctln` *cmTransportLedgerName*	Specifies the TibrvCmTransport ledger file.

-tctsl *cmTransportSyncLedger*	Specifies if the endpoint uses a synchronous ledger. Valid values are `true` or `false`.
-tctra *cmTransportRelayAgent*	Specifies the endpoint's TibrvCmTransport relay agent.
-tctdtl *cmTransportDefaultTimeLimit*	Specifies the default time limit for a Certified Message to be delivered.
-tclca *cmListenerCancelAgreements*	Specifies if Certified Message agreements are canceled when the endpoint disconnects. Valid values are `true` or `false`.
-tcqtsn *cmQueueTransportServerName*	Specifies the server's TibrvCmQueueTransport correspondent name.
-tcqtcn *cmQueueTransportClientName*	Specifies the client's TibrvCmQueueTransport correspondent name.
-tcqtww *cmQueueTransportWorkerWeight*	Specifies the endpoint's TibrvCmQueueTransport `worker weight`.
-tcqtws *cmQueueTransportWorkerTasks*	Specifies the endpoint's TibrvCmQueueTransport `worker tasks` parameter.
-tcqtsw *cmQueueTransportSchedulerWeight*	Specifies the TibrvCmQueueTransport `scheduler weight` parameter.
-tcqtsh *cmQueueTransportSchedulerHeartbeat*	Specifies the endpoint's TibrvCmQueueTransport `scheduler heartbeat` parameter.
-tcqtsa *cmQueueTransportSchedulerActivation*	Specifies the TibrvCmQueueTransport `scheduler activation` parameter.
-tcqtct *cmQueueTransportCompleteTime*	Specifies the TibrvCmQueueTransport `complete time` parameter.
-tmnfv *messageNameFieldValue*	
-tmnfp *messageNameFieldPath*	
-tbfi *bindingFieldId*	
-tbfn *bindingFieldName*	

`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *`dir`*	Specifies the output directory for the generated contract.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Example

Example 98 shows an Artix description for a TIB/RV port.

Example 98: *TIB/RV Port Description*

```
<service name="BaseService">
    <port binding="tns:BasePortBinding" name="BasePort">
        <tibrv:port serverSubject="Artix.BaseService.BasePort"/>
    </port>
</service>
```

More information

For a complete listing of the attribute used in configuring a TIB/RV service see “Tibco Port Properties” on page 533.

Creating a Tuxedo Service

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 port using Tuxedo in the physical part of an Artix contract. The extensions used to describe a Tuxedo port are defined in the following namespace:

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

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

Defining the Tuxedo services

As with other transports, the Tuxedo transport description is contained within a `port` element. Artix uses `tuxedo:server` to describe the attributes of a Tuxedo port. `tuxedo:server` has a child element, `tuxedo:service`, that gives the bulletin board name of a Tuxedo port. The bulletin board name for the service 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 port 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 Artix Designer

To add a Tuxedo port from the Artix Designer Diagram view:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service** to bring up the **Select Source Resources** window.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.

4. Click **Next** to bring up the **Define Service** window.
5. Enter a name for the new service in the **Name** field.
6. Click **Next** to bring up the **Define Port** window.
7. Enter a name for the new port in the **Name** field.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next** to bring up the **Define Port Properties** window.
10. Select **tuxedo** from the **Transport Type** drop-down list to bring up the Tuxedo transport properties editor shown in Figure 72.

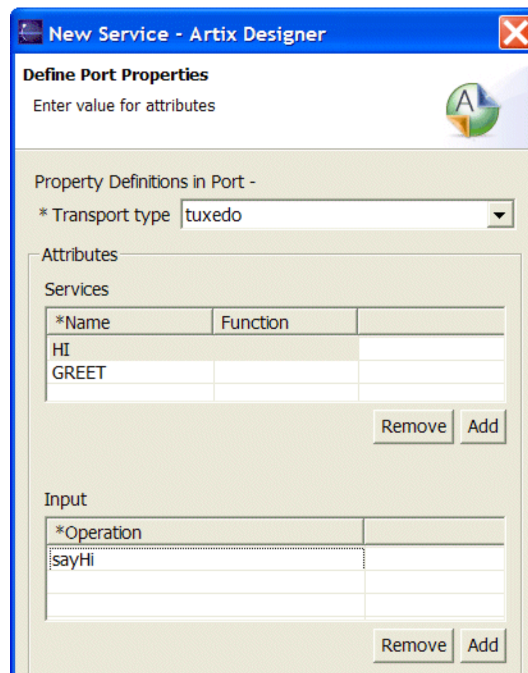


Figure 72: *Editing Tuxedo Transport Attributes*

11. Click the **Add** button under the **Services** table to add a Tuxedo service to the table.
12. Click in the **Name** column to edit the service's name.

13. Add an optional value for the `function` attribute in the **Function** column.
14. With the service selected in the **Service** table, click the **Add** button under the **Operations** table.
15. Select one of the operations from the window that pops up.
16. Click **OK** to return to editing the transport properties.
17. Repeat steps **14** through **16** until you have added all of the desired operations for the service.
18. Repeat steps **11** through **17** until you have added all of the desired Tuxedo services to the port.
19. Click **Finish**.

Using the command line tools

You can use the `wsdltoservice` command line tool to add a Tuxedo service definition to an Artix contract. To use `wsdltoservice` to add a Tuxedo service 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] [-q] [-h] [-V]  wsdlurl
```

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

`-e` *service*	Specifies the name of the generated service.
`-t` *port*	Specifies the value of the `name` attribute of the generated `port` element.
`-b` *binding*	Specifies the name of the binding for which the service is generated.
`-tsn` *tuxService*	Specifies the name of the Tuxedo bulletin board to which Artix connects.
`-tsn` *tuxService*:*tuxFunction*	Specifies the name of the function to be used on the specified Tuxedo bulletin board.
`-ton` *tuxService*:*operation*	Specifies the WSDL operation that is handled by the specified Tuxedo endpoint.

`-o` *`file`*	Specifies the filename for the generated contract. The default is to append `-service` to the name of the imported contract.
`-d` *`dir`*	Specifies the output directory for the generated contract.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Example

An Artix contract exposing the `personalInfoService` as a Tuxedo service would contain a `service` element similar to Example 99 on page 308.

Example 99: *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>
```

Configuring a Service to Use Codeset Conversion

Overview

While many of the bindings support 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 programatically using the context mechanism. The programatic 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 *Deploying and Managing 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 100 show 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 100:*Specifying Codeset Conversion*

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

Using the endpoint definition above, the client will convert its requests into UTF-8 before sending them to the server. The server will convert its replies into ISO-8859-1 before sending them to the client. The client's 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.

CHAPTER 10

Adding Routing Instructions

Artix provides messages routing based on operations, ports, or message attributes.

In this chapter

This chapter discusses the following topics:

Artix Routing	page 314
Compatibility of Ports and Operations	page 315
Defining Routes in Artix Contracts	page 318
Creating Routes Using Artix Designer	page 330
Creating Routes from the Command Line	page 334
Load Balancing	page 338
Error Handling	page 339
Service Lifecycles	page 340
Routing References to Transient Servants	page 342

Artix Routing

Overview

Artix routing is implemented within an Artix service and is controlled by rules specified in the service's contract. Artix services that include routing rules can be deployed either in standalone mode or embedded into an Artix service.

Artix supports the following types of routing:

- Port-based
- Operation-based

A router's contract must include definitions for the source services and destination services. The contract also defines the routes that connect source and destination ports. This routing information is all that is required to implement port-based or operation-based routing.

Port-based

Port-based routing acts on the port or transport-level identifier, specified by a `port` element in an Artix contract. This is the most efficient form of routing. Port-based routing can also make a routing decision based on port properties, such as the message header or message identifier. Thus Artix can route messages based on the origin of a message or service request, or based on the message header or identifier.

Operation-based

Operation-based routing lets you route messages based on the logical operations described in an Artix contract. Messages can be routed between operations whose arguments are equivalent. Operation-based routing can be specified on the interface, `portType`, level or the finer grained operation level.

Compatibility of Ports and Operations

Overview

Artix can route messages between services that expect similar messages. The services 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 may 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 have an operation that uses messages with the same logical description in the Artix contract defining their integration. For port-based routing, the destination service must have a matching operation defined for each of the operations defined for the source service.

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 operations defined in the logical interface, or `portType`, to which the port is bound. Therefore, port-based routing requires that the services between which messages are being routed must have compatible logical interface descriptions.

For two ports to have compatible logical interfaces the following conditions must be met:

- The destination's logical interface must contain a matching operation for each operation in the source's logical interface. Matching operations must have the same name.
- Each of the matching operations must have the same number of input, output, and fault messages.
- Each of the matching operations' messages must have the same sequence of part types.

For example, given the two logical interfaces defined in Example 101 you could construct a route from a port bound to `baseballScorePortType` to a port bound to `baseballGamePortType`. However, you could not create a

route from a port bound to `finalScorePortType` to a port bound to `baseballGamePortType` because the message types used for the `getScore` operation do not match.

Example 101:*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:basballScore" 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:basballScore" name="baseballScore"/>
  </operation>
</portType>
<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 provides a finer grained level of control over how messages can be routed. Operation-based routing rules check for compatibility on the `operation` level of the logical interface description. Therefore, messages can be routed between any two compatible messages.

The following conditions must be met for operations to be compatible:

- The operations must have the same number of input, output, and fault messages.
- The messages must have the same sequence of part types.

For example, if you added the logical interface in Example 102 to the interfaces in Example 101 on page 316, 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 102:*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 port-based and operation-based routing are fully implemented in the contract defining the integration of your systems. Routes are defined using WSDL extensions that are defined in the namespace `http://schemas.iona.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 serves as the source for messages that will be routed using the route.

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 319
Using Operation-Based Routing	page 322
Advanced Routing Features	page 325

Using Port-Based Routing

Overview

Port-based routing is the highest performance type of routing Artix performs. It is also the easiest to implement. All of the rules are specified in the Artix contract describing how your systems are integrated. The routes specify the source port for the messages and the destination port 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:

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

To describe a port-based route you use three elements:

routing:route

`routing:route` 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. `route` also has an optional attribute, `multiRoute`, which is discussed in "Advanced Routing Features" on page 325.

routing:source

`routing:source` specifies the port from which the route will redirect messages. A route can have several source elements as long as they all meet the compatibility rules for port-based routing discussed in "Port-based routing" on page 315.

`routing:source` requires two attributes, `service` and `port`. `service` specifies the service element in which the source port is defined. `port` specifies the name of the port element from which messages are being received. The router will create a proxy for to listen for mesages on this port.

routing:destination

`routing:destination` specifies the port to which the source messages are directed. The destination must be compatible with all of the source elements. For a discussion of the compatibility rules for port-based routing see “Port-based routing” on page 315.

In standard routing only one destination is allowed per route. Multiple destinations are allowed in conjunction with the route element’s `muliRoute` attribute that is discussed in “Advanced Routing Features” on page 325.

`routing:destination` requires two attributes, `service` and `port`. `service` specifies the service element in which the destination port is defined. `port` specifies the name of the port element to which messages are being sent.

Example

For example, to define a route from `baseballScorePortType` to `baseballGamePortType`, defined in Example 101 on page 316, your Artix contract would contain the elements in Example 103.

Example 103:*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 103:

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 operations within a logical interface as a source or a destination.

Like port-based routing, operation-based routing is fully implemented by adding routing rules to Artix contracts.

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 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>`. `<routing:operation>` specifies an operation defined in the source port's logical interface and an optional target operation in the destination port's logical interface. You can specify any number of operation elements in a route. The operation elements must be specified after all of the source elements and before any destination elements.

`operation` takes one required attribute, `name`, that specifies the name of the operation in the source port's logical interface that is to be used in the route.

`operation` also has an optional attribute, `target`, that specifies the name operation in the destination port's logical interface to which the message is to be sent. 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 317.

How operation-based rules are applied

Operation-based routing rules apply to all of the source elements listed 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 ports specified in a source element.

- The operation name associated with the received message is specified in one of the `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 102 on page 317, to `getScore` defined in `finalScorePortType`, shown in Example 101 on page 316, your Artix contract would contain the elements in Example 104.

Example 104: *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="finalScoreSerice">
    <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 104:

1. The logical interfaces must be bound to physical ports in `service` elements of the Artix contract.
2. The route, `scoreRoute`, is defined using the `<route:operation>` element.

You could also create a route between `getScore` in `baseballGamePortType` to a port bound to `baseballScorePortType`; see Example 101 on page 316.The resulting contract would include the fragment shown in Example 105.

Example 105:*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 315.

Advanced Routing Features

Overview

Artix routing also supports the following advanced routing capabilities:

- Broadcasting a message to a number of destinations.
- Specifying a failover service to route messages to provide a level of high-availability.
- Routing messages based on transport attributes in the received message's header.

Message broadcasting

Broadcasting a message with Artix is controlled by the routing rules in an Artix contract. Setting the `multiRoute` attribute to the `<routing:route>` element to `fanout` in your route definition allows you to specify multiple destinations in your route definition to which the source messages are broadcast.

There are three restrictions to using the fanout method of message broadcasting:

- All of the sources and destinations must be oneways. In other words, they cannot have any output messages.
- The sources and destinations cannot have any fault messages.
- The input messages of the sources and destinations must meet the compatibility requirements as described in "Compatibility of Ports and Operations" on page 315.

Example 106 shows an Artix contract fragment describing a route for broadcasting a message to a number of ports.

Example 106: *Fanout Broadcasting*

```
<message name="statusAlert">
  <part name="alertType" type="xsd:int"/>
  <part name="alertText" type="xsd:string"/>
</message>
<portType name="statusGenerator">
  <operation name="eventHappens">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
```

Example 106:*Fanout Broadcasting*

```
<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

Artix failover routing is also specified using the `routing:route`'s `multiRoute` attribute. To define a failover route you set `multiRoute` to equal `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.

Given the route shown in Example 107, the message will first be routed to `destinationPortA`. If service on `destinationPortA` cannot receive the message, it is routed to `destinationPortB`.

Example 107: *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.

Routing based on transport attributes

Artix allows you to specify routing rules based on the transport attributes set in a message's header when using HTTP or WebSphere MQ. 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 to the `<routing:tranportAttribute>`. 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 `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 25.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 25: *Context QNames*

Context QName	Details
`http-conf:HTTPServerIncomingContexts`	Contains the attributes for HTTP messages being received by a server.
`corba:corba_input_attributes`	Contains the data stored in the CORBA principle
`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. Attributes dealing with string comparisons 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. `contains` accepts an 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 "HTTP Port Properties" on page 451. For information on the transport attributes for WebSphere MQ see "WebSphere MQ Port Properties" on page 493.

Example 108 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 108: *Transport Attribute Rules*

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

Creating Routes Using Artix Designer

Overview

Artix Designer includes a routing wizard that assists you in creating routes from the services available in your contract. It walks you through the steps of creating a route and provides you with the valid options for the services available. It performs all of the compatibility testing for you and will never allow you to create an invalid route.

Using Artix Designer

To create a route from the Artix Designer Diagram view:

1. Right-click the **Routes** node to activate the pop-up menu.
2. Select **New Route** to bring up the **Select Source Resources** window.
3. Select at least one contract from the list to act as a source of services between which to route.

   All of the services defined in the selected contracts will be made available for you to use in defining a route. The contracts will be imported to contract using WSDL `import` elements.

4. Click **Next** to bring up the **Define Endpoints** window, shown in Figure 73.

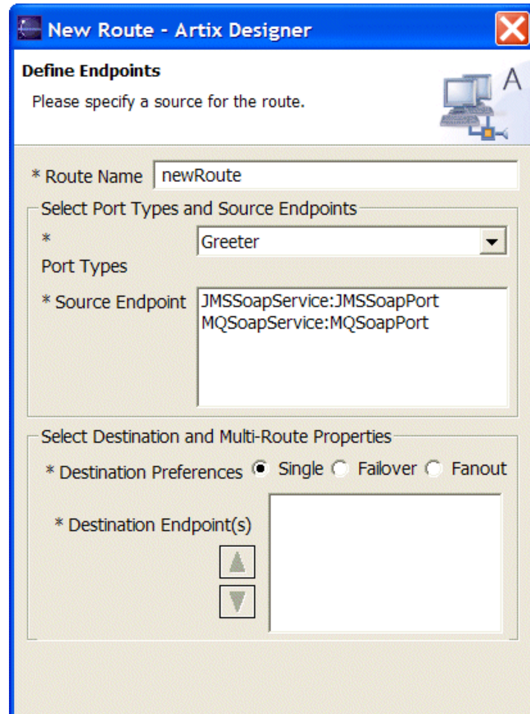


Figure 73: *Defining the Endpoints of a Route*

5. Enter a name for the new route in the **Route Name** field.
6. Select the interface that bound to the service that will be the source endpoint for the route from the **Port Type** drop-down list.
7. Select one service from the **Source Endpoint** table to be the source endpoint for the route.
8. Select one of the routing styles from **Destination Preferences**.
    - **Single** specifies that the route will be a standard point to point route.

- **Failover** specifies that the route will be a multipoint route that will be used to redirect messages when one of the destination endpoints fails. See "Failover routing" on page 326.
- **Fanout** specifies that the route will be a multipoint route that broadcasts messages to all of the selected endpoints. See "Message broadcasting" on page 325.
- **Load Balance** specifies that the route will be used to load balance among all of the selected destination endpoints. See "Load Balancing" on page 338.

9. Select the endpoints that you want to be endpoints from the **Destination Endpoints** table.

> **Note:** When the route is created, the destination endpoints are placed according to their position in the **Destination Endpoint** table. For failover and load balancing routes, you can use the arrows to the left of the table to rearrange the order of the endpoints in the table.

10. Click **Next** to bring up the **Specify Operations** window, shown in Figure 74.

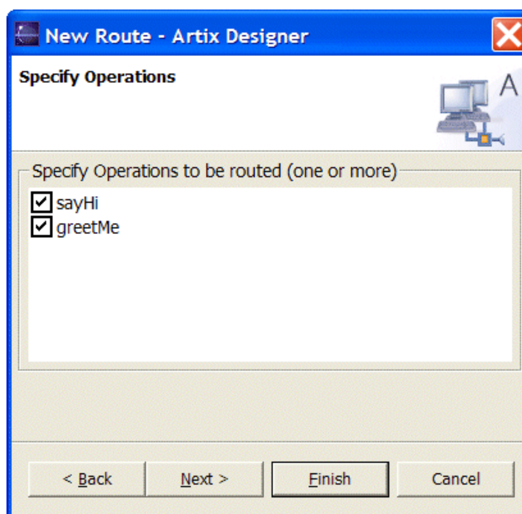


Figure 74: *Selecting the Operations to Use for the Route*

11. Select at least one operation to use in the route.
12. Click **Next** to bring up the **Set Transport Attributes** window, shown in Figure 75.

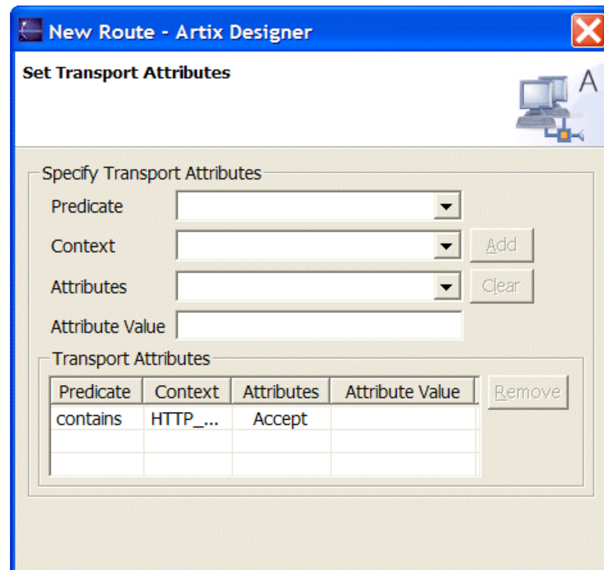


Figure 75: *Specifying Transport Attributes to Constrain a Route*

13. Define an attribute routing rule. See "Routing based on transport attributes" on page 327.
14. Click **Add** to add the rule to the Transport Attributes table.
15. Repeat steps **13** and **14** until you have added all of the desired attribute routing rules.
16. Click **Finish**.

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 and destination, 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] [-?] [-v]
              [-verbose] [-L file] [-q] [-h] [-V]  wsdlurl
```

`wsdltorouting` has the following options.

`-rn` *name*	Specifies the name of the generated route. If no name is given a unique name will be generated for the route.
`-ssn` *service*	Specifies the name of the service to use as the source of the route.
`-spn` *port*	Specifies the name of the port to use as the source of the route. The port must correspond to a `port` element in the specified service.
`-dsn` *service*	Specifies the name of the service to use as the destination of the route.
`-dpn` *port*	Specifies the name of the port to use as the destination of the route. The port must correspond to a `port` element in the specified service.
`-on` *operation*	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.
`-ta` *attribute*	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 335.
`-d` *dir*	Specifies the output directory for the generated contract.

`-o file`	Specifies the filename of the generated contract.
`-?`	Displays the tool's usage statement.
`-v`	Displays the tool's version.
`-verbose`	Turns on verbose mode.
`-L file`	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-V`	Specifies that the tool runs in verbose mode.

Specifying transport attributes

When using `wsdltorouting`, 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 26 shows the valid context names to use in specifying a transport attribute.

Table 26: *Context Names Used with wsdltorouting*

Context Name	Artix Context
`HTTP_SERVER_INCOMING_CONTEXTS`	HTTP properties received as part of a client request
`CORBA_CONTEXT_ATTRIBUTES`	CORBA transport properties
`SECURITY_SERVER_CONTEXT`	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 27 shows the valid condition entries used in specifying transport attributes when using `wsdltorouting`.

Table 27: *Conditions Used with wsdltorouting*

Condition	WSDL Equivalent
`equals`	routing:equals
`startswith`	routing:startswith
`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 109 to add a route to your contract.

Example 109:*Adding a Route with wsdltorouting*

```
wsdltorouting -rn itchyGougeScratchy -ssn itchy -spn gougerPort
        -dsn scratchy -dpn gougedPort -on gouge
       -ta HTTP_SERVER_INCOMING_CONTEXTS,UserName,equals,Goering
        itchyscratchy.wsdl
```

The resulting route is shown in Example 110.

Example 110: *Route from wsdltorouting*

```
<routing:route name="itchyGougeScratchy">
  <routing:source service="tns:itchy"
                  port="tns:gougerPort"/>
  <routing:operation name="gouge"/>
  <routing:trasnportAttributes>
    <rotuing:equals
              contextName="http-conf:HTTPServerIncomingContexts"
              contextAttributeName="UserName"
              value="Goering"/>
  </routing:transportAttributes>
  <routing:destination service="tns:scratchy"
                       port="gougedPort"/>
</routing:route>
```

Load Balancing

Overview

The router can load balance requests across a number of servers 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 services.

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 `multiRoute` to equal `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 services 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 111.

Example 111: *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 service 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.

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 services that have patterns such as callback, factory, or any interaction that passes references to other services. 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 lifecycle 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 may attempt to proxify to a service 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 may not be possible if the service 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 a service is no longer needed when it is garbage collected. However, if you set `max_cache_size` too high, this may 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 defined in your contract. The cloned service uses the same interface, binding, and transport as the service defined in the contract. However, it has a unique QName and a unique address. So, when a transient servant's service definition only exists in the memory of the application that created it and possesses no link back to the service from which it was cloned.

Because a transient servant does not have a service definition 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 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 definition in its contract.

Compatibility of services

A service is considered compatible with a transient servant if it uses the same interface, binding, and transport as the transient servant. For example, if transient servant was created using the `templateVendor` service defined in Example 112 it would be compatible with `IIOPVendor`. However, it would not be compatible with `SOAPVendor` because `SOAPVendor` uses a different transport than template. 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 112:*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 112: *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="lcoalhost: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 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 services through the router, the potential exists for the router to select the static service 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 definition.

To avoid this situation be sure to place the service templates used to create transient servants before the service definitions that will be used to create static servants. This will ensure that the router will find the service templates first when it proxifies a reference to a transient servant.

Fastrack Service Enabling

The Artix Designer offers fastrack paths to Web and CORBA service enabling.

In this chapter

This chapter discusses the following topics:

Web Service Enabling a Service	page 346
CORBA Enabling a Service	page 348

Web Service Enabling a Service

Overview

If you have a contract with a fully defined interface and you know that it is going to be exposed as a Web service using SOAP/HTTP, then you can use the Artix Designer's **SOAP Enable** menu option.

This option automatically adds a default SOAP binding and a SOAP port to your contract.

You provide the name of the interface from which to generate the binding, how you want the SOAP binding configured, the address of the HTTP port the service will be exposed on, and the names for the binding and the service. The Artix Designer does the rest.

Procedure

To Web service enable an interface using **SOAP Enable**:

1. Right-click the contract defining the interface you want to Web service enable to activate the pop-up menu.
2. Select **Artix|SOAP Enable** to bring up the **SOAP Binding and Service Details** window, shown in Figure 76.

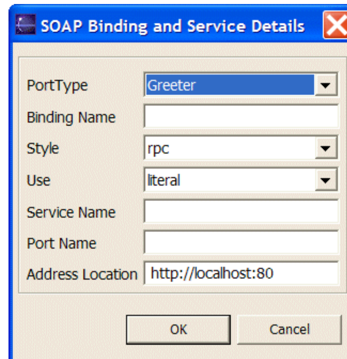


Figure 76: *SOAP Binding and Service Details Window*

3. Select the interface you want to enable from the **PortType** drop-down list.

4. Enter a name for the generated binding in the **Binding Name** field.
5. Select the SOAP style, `rpc` or `doc`, from the **Style** drop-down list.
6. Select the SOAP usage, `literal` or `encoded`, from the **Use** drop-down list.
7. Enter a name for the generated service in the **Service Name** field.
8. Enter a name for the generated port in the **Port Name** field.
9. Enter the HTTP address of the new service in the **Address Location** field.
10. Click **OK**.
11. Save the contract.

CORBA Enabling a Service

Overview

If you have a contract with a fully defined interface and you know that it is going to be exposed as a CORBA service, then you can use the Artix Designer's **CORBA Enable** menu option.

This option automatically adds a CORBA binding and a CORBA port to your contract.

You provide the name of the interface from which to generate the binding, the name for the binding, and the name for the service. The generated `port` element will need to be edited to have a valid IOR.

Procedure

To CORBA enable an interface using **CORBA Enable**:

1. Right-click the contract defining the interface you want to CORBA enable to activate the pop-up menu.
2. Select **Artix|CORBA Enable** to bring up the **CORBA Binding and Service Details** window, shown in Figure 77.

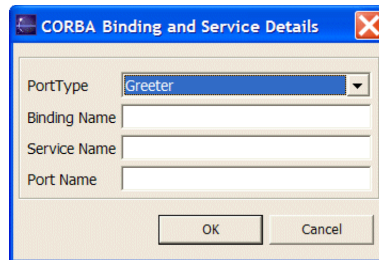


Figure 77: *CORBA Binding and Service Details Window*

3. Select the interface you want to enable from the **PortType** drop-down list.
4. Enter a name for the generated binding in the **Binding Name** field.
5. Enter a name for the generated service in the **Service Name** field.
6. Enter a name for the generated port in the **Port Name** field.
7. Click **OK**.

8. In the contract, edit the `corba:address` element in the port definition to have a valid IOR.
9. Save the contract.

CHAPTER 12

Editing Artix Resources

The Artix Designer makes it easy to edit the resources in your Artix project.

In this chapter

This chapter discusses the following topics:

Editing Contracts and Schemas	page 352
Editing Generated Resources	page 367

Editing Contracts and Schemas

Overview

The Artix Designer provides two views for editing your contracts and schema documents. *Diagram* view shows a graphical representation of the elements in the file and provides you with wizards that allow you to edit the elements. *Source* view shows the file's XML source and allows you to manually edit it.

In this section

This section discusses the following topics:

Working with the Editor Views	page 353
Editing Types	page 356
Editing Messages	page 357
Editing Port Types	page 359
Editing Bindings	page 363
Editing Services	page 364
Editing Routes	page 365

Working with the Editor Views

Overview

The Artix Designer's WSDL editor contains two views:

- Diagram view
- Source view

Editing in Diagram view

The Diagram view, shown in Figure 78, is selected by clicking the **Diagram** tab at the bottom of the WSDL editor view.

The Diagram view shows all of the sections of a contract. Sections that have entries are expandable. Empty sections, such as the Services and Routes sections in Figure 78 are not.

To edit an entry, right-click the element to bring up the pop-up menu. To add new elements, right-click a root element.

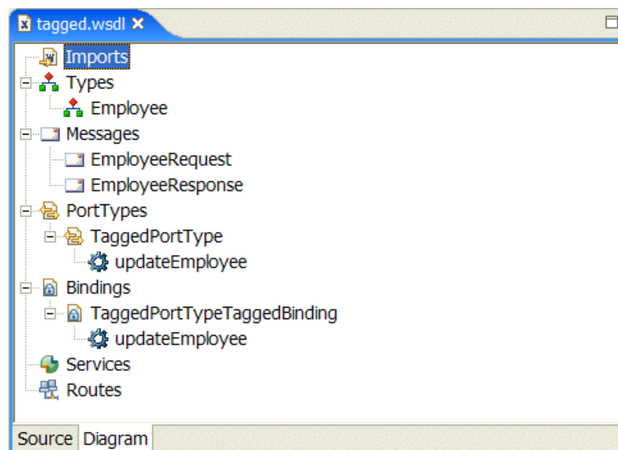


Figure 78: *Editing in Diagram View*

Editing in Source view

The Source view, shown in Figure 79, is selected by clicking the **Source** tab at the bottom of the WSDL editor view.

Figure 79: *Editing in Source View*

This view displays the file in its native XML format. You can use it to directly modify the document's XML source. This can be useful if you want to add a type that is not supported by the Artix Designer wizards, or if you want just make a quick change to a file.

To navigate through a file's XML, you can select elements from the **Outline** view, shown in Figure 80.

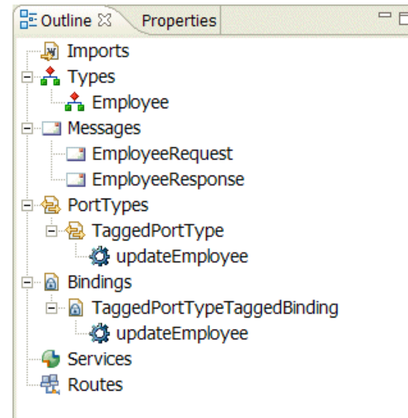


Figure 80: *The Outline View*

This will bring you to the element's XML root in the WSDL editor Source view. For example, selecting **Employee** from the **Outline** view, will cause the cursor to jump to the `<complexType name="Employee">` element in the WSDL source.

Before saving a WSDL file, Artix will validate the WSDL. If it contains any errors, they will be displayed in the **Problems** window and the **Output** window will show the message **Invalid WSDL - no data available**. The error messages will tell you which line in the WSDL appears to be invalid.

Editing Types

Overview

Types define particles of data that are used to build the messages exchanged by services. A type can be made up of smaller pieces that can be combined in a variety of ways. For more information on types, see "Defining Data Types" on page 91.

When editing types in diagram view you have the following options:

- Renaming a type
- Deleting a type

Renaming a type

To rename a type from diagram view:

1. Select the type you want to rename from the diagram.
2. Right-click the type to activate the pop-up menu.
3. Select **Rename Type** from the pop-up menu to bring up the **Rename Component** window.
4. Enter a new name for the type in the **New Name** field.
5. Click **OK**.

Deleting a type

To delete a type from diagram view:

1. Select the type you want to delete from the diagram.
2. Right-click the type to activate the pop-up menu.
3. Select **Delete Type** from the pop-up menu to bring up the **Confirm Deletion** window.
4. Click **Yes**.

Editing Messages

Overview

A message defines a block of data that is used by a service. It consists of one or many parts. Each part in the message represents a discreet piece of data that is either a native XML Schema type, or a type defined in the contract. For more information on messages see “Defining Messages” on page 123.

In diagram view you have the following options available for editing a message:

- Changing a message
- Renaming a message
- Deleting a message

Changing a message

To edit a message in diagram view:

1. Select the message you want to edit from the diagram.
2. Right-click the message to activate the pop-up menu.
3. Select **Edit Message** from the pop-up menu to bring up the **Define Message Properties** window.
4. If you want to change the name of the message, enter a new name in the **Name** field.
5. Click **Next** to open the **Define Message Parts** window.
6. If you want to edit one of the existing message parts:
   i. Select the message part from the **Part List**.
   ii. The message will appear in the fields above the list and the **Add** button will change to **Update**.
   iii. To change the name of the message part, enter a new name in the **Name** field.
   iv. To change the data type of the message part, select a new data type from the **Type** drop-down list.
   v. Click **Update** to make your changes to the message part.

7. To add a new part to the message:
   i. Enter a name for the message part in the **Name** field.
   ii. Select a data type for the message part from the **Type** drop-down list.
   iii. Click **Add**.
8. Repeat steps **6** and **7** until you finished editing the message.
9. Click **Finish** to write the message changes to the contract.

Renaming a message

To rename a message from diagram view:

1. Select the message you want to rename from the diagram.
2. Right-click the message to activate the pop-up menu.
3. Select **Rename Message** from the pop-up menu to bring up the **Rename Component** window.
4. Enter a new name for the message in the **New Name** field.
5. Click **OK**.

Deleting a message

To delete a message from diagram view:

1. Select the message you want to delete from the diagram.
2. Right-click the message to activate the pop-up menu.
3. Select **Delete Message** from the pop-up menu to bring up the **Confirm Deletion** window.
4. Click **Yes**.

Editing Port Types

Overview

A port type defines a service's interface. It is a collection of operations provided by a service. The operations defined inside of the port type are defined by specifying the messages that are exchanged when the operation is evoked. When Artix generates code from a contract, the port types are translated into objects and the operations are mapped into methods for the generated objects. For more information on port types see "Defining Your Interfaces" on page 127.

In diagram view you have the following options available for editing a port type:

- Adding a new operation
- Renaming a port type
- Deleting a port type
- Editing an operation
- Renaming an operation
- Deleting an operation

Adding a new operation

To add a new operation to a port type:

1. Select the port type you to which you want add the new operation.
2. Right-click the port type to activate the pop-up menu.
3. Select **New Operation** from the pop-up menu to bring up the **Define Port Type Operations** window.
4. Enter a name for the new operation in the **Name** field.
5. Select an operation style from the **Style** drop-down list.

   Operations can have one of the following styles:

   - **One-way** operations have only an input message. They cannot return any data to the client.
   - **Request-response** operations have an input message, an output message, and any number of optional fault messages.
6. Click **Next** to open the **Define Operation Messages** window.
7. Select a message type for the new operation message from the **Type** drop-down list.

Operation messages can be of one of the following types:

- **input** messages represent data that a client send to the server.
- **output** messages represent data that a service returns to a client.
- **fault** messages represent data that a service returns to a client in the event that an error occurred while processing the request.

> **Note:** If your operation is oneway, input will be the only message type available.

8. Select the global message that defines the data passed by this operation message from the **Message** drop-down list.
9. Enter a name for the operation message in the **Name** field.
10. Click **Add** to add the message to the **Operation Messages** table.
11. Repeat steps **7** through **10** until all of the operational messages have been specified.
12. Click **Finish**.

Renaming a port type

To rename a port type from diagram view:

1. Select the port type you want to rename from the diagram.
2. Right-click the port type to activate the pop-up menu.
3. Select **Rename Port Type** from the pop-up menu to bring up the **Rename Component** window.
4. Enter a new name for the port type in the **New Name** field.
5. Click **OK**.

Deleting a port type

To delete a port type from diagram view:

1. Right-click the desired port type to activate the pop-up menu.
2. Select **Delete Port Type** from the pop-up menu to open the **Confirm Deletion** widow.
3. Select **Yes**.

Editing an operation

To edit an operation from diagram view:

1. Right-click the desired operation to activate the pop-up menu.
2. Select **Edit Operation** from the pop-up menu to bring up the **Define Port Type Operations** window.
3. To change the name of the operation, enter a new name in the **Name** field.
4. To change the style of the operation, select a new operation style from the **Style** drop-down list.
5. Click **Next** to open the **Define Operation Messages** window.
6. To edit an existing operation message:
   i. Select a message from the **Operation Messages** table.
   ii. The details of the message will appear in the **Messages for Port Type Operation** fields and the **Add** button will change to **Update**.
   iii. To change the message's type, select a new type from the **Type** drop-down list.
   iv. To change the global message associated with the operation message, select a new global message from the **Message** drop-down list.
   v. To change the message's name, enter a new name in the **Name** field.
   vi. To save the changes, click **Update**.
7. To add a new operation message:
   i. Select a message type for the new operation message from the **Type** drop-down list.
   ii. Select the global message that defines the data passed by this operation message from the **Message** drop-down list.
   iii. Enter a name for the operation message in the **Name** field.
   iv. Click **Add** to add the message to the **Operation Messages** table.
8. Repeat steps 6 and 7 until you have made all of your changes.
9. Click **Finish**.

Renaming an operation

To rename an operation from diagram view:

1. Right-click the desired operation to activate the pop-up menu.
2. Select **Rename Operation** from the pop-up menu to bring up the **Rename Component** window.
3. Enter a new name for the operation in the **New Name** field.
4. Click **OK**.

Deleting an operation

To delete an operation from diagram view:

1. Right-click the desired operation to activate the pop-up menu.
2. Select **Delete Operation** from the pop-up menu to open the **Confirm Deletion** window.
3. Select **Yes**.

Editing Bindings

Overview

Bindings map the abstract data used to define operations into a concrete data format that can be passed over a network. For more information on bindings, see "Binding Interfaces to a Payload Format" on page 135.

When editing bindings in diagram view you have the following options:

- Renaming a binding
- Deleting a binding

Renaming a binding

To rename a binding from diagram view:

1. Right-click the desired binding to activate the pop-up menu.
2. Select **Rename Binding** from the pop-up menu to bring up the **Rename Component** window.
3. Enter a new name for the binding in the **New Name** field.
4. Click **OK**.

Deleting a binding

To delete a binding from diagram view:

1. Right-click the desired binding to activate the pop-up menu.
2. Select **Delete Binding** from the pop-up menu to bring up the **Confirm Deletion** window.
3. Click **Yes**.

Editing Services

Overview

Services define the network interface over which a service is exposed. For more information on services, see “Adding Transports” on page 251.

When editing services in diagram view you have the following options:

- Renaming a service
- Deleting a service

Renaming a service

To rename a service from diagram view:

1. Right-click the desired service to activate the pop-up menu.
2. Select **Rename Service** from the pop-up menu to bring up the **Rename Component** window.
3. Enter a new name for the service in the **New Name** field.
4. Click **OK**.

Deleting a service

To delete a service from diagram view:

1. Right-click the desired service to activate the pop-up menu.
2. Select **Delete Binding** from the pop-up menu to bring up the **Confirm Deletion** window.
3. Click **Yes**.

Editing Routes

Overview

Routes define how different systems with similar endpoints are integrated. They can also be used for load balancing. For more information on routes, see “Adding Routing Instructions” on page 313.

When editing routes in diagram view you have the following options:

- Editing a route
- Renaming a route
- Deleting a route

Editing a route

To edit a route from diagram view:

1. Right-click the desired route to activate the pop-up menu.
2. Select **Edit Route** from the pop-up menu to bring up the **Define Endpoints** window.
3. To change the route’s name, enter a new name in the **Route Name** field.
4. To change the port type that defines the source endpoint, select a new port type from the **Port Types** drop-down list.
5. To change the source endpoint, select a new endpoint from the **Source Endpoint** table.
6. To change the routes destination change the settings under **Select Destination and Multi-Route Properties**. See “Adding Routing Instructions” on page 313.
7. Click **Next** to bring up the Specify Operations windowSelect the operation for which you want to this route to apply.
8. Click **Next** to bring up the **Set Transport Attributes** window
9. To change an existing transport attribute rule:
   i. Select a transport attribute from the **Transport Attributes** table.
   ii. The details of the rule will appear in the **Specify Transport Attributes** fields and the **Add** button will change to **Update**.
   iii. To change the rule’s predicate, select a new predicate from the **Predicate** drop-down list.

iv. To change the transport attribute context on which the rule is based, select a new transport context from the **Context** drop-down list.

v. To change the attribute which the rule evaluates, select a new attribute from the **Attributes** drop-down list.

vi. To change the attribute value on which the rule matches, enter a new value in the **Attribute Value** field.

vii. To save the changes, click **Update**.

10. To add a new transport attribute rule:

    i. Select a predicate for the new rule from the **Predicate** drop-down list.

    ii. Select the transport attribute context on which to base the rule from the **Context** drop-down list.

    iii. Select the attribute to be evaluated by the rule from the **Attributes** drop-down list.

    iv. Enter a the value on which the rule matches into the **Attribute Value** field.

    v. Click **Add** to add the rule to the **Transport Attributes** table.

11. Repeat steps **9** and **10** until all of your changes are complete.

12. Click **Finish** to save the changes to the contract.

Renaming a route

To rename a route from diagram view:

1. Right-click the desired route to activate the pop-up menu.
2. Select **Rename Route** from the pop-up menu to bring up the **Rename Component** window.
3. Enter a new name for the route in the **New Name** field.
4. Click **OK**.

Deleting a route

To delete a route from diagram view:

1. Right-click the desired route to activate the pop-up menu.
2. Select **Delete Route** from the pop-up menu to bring up the **Confirm Deletion** window.
3. Click **Yes**.

Editing Generated Resources

Overview

Generated resources include C++ code, Java code, configuration domains, and deployment scripts. These resources are the physical implementation and the deployment details of the services defined in the contracts and XML Schema documents in a project.

You specify what resources are generated for a project by defining generation profiles. A project can have multiple generation profiles. Each profile will be responsible for generating the implementation and the deployment details for a different piece of an overall solution.

Editing generated resources

All of the generated resources are editable using the built in text and code edit facilities of the Eclipse platform. For example, if you need to edit a configuration domain, Eclipse will open the file in a text editor. However, if you edit a Java class, Eclipse will open the file in its Java editor.

CHAPTER 13

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 an Artix Server	page 370
Using Artix to Facilitate Interface Versioning	page 372
WSDL Messages and the Transformer	page 377
Writing XSLT Scripts	page 380

Using the Artix Transformer as an Artix Server

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 an Artix server 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 server 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 "Binding Interfaces to a Payload Format" on page 135. For information on adding network details for the transformer read "Adding Transports" on page 251.

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 380.

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 an Artix server. The first is to configure the transformer to load in its own process using the Artix Standalone Service. 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 Deploying and Managing 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 programing.

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 113.

Example 113: *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 113 on page 373 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 114.

Example 114: *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 "Binding Interfaces to a Payload Format" on page 135. For information on adding network details for the transformer read "Adding Transports" on page 251.

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 380.

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 Deploying and Managing 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 of the transformer in an instance of the Artix Standalone service.

For a detailed discussion of how to configure and deploy the Artix Chain Builder see Deploying and Managing 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 root element of the incoming message is taken from the `name` attribiute of the operation's `input` message.
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 115 and the transformer implemented the operation `configure` the XML document would be constructed using the message `oldClientInput`, which is the `input` message.

Example 115: *WSDL Fragment for Transformer*

```
<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 115: *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="original"/>
    <output name="updatedInput" message="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 115 on page 377 an input message processed by the transformer could look like Example 116.

Example 116: *Transformer Input Message*

```
<oldClientInput>
  <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 root element of the result has the name of the output message, as defined in 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 115 on page 377 would look like Example 117.

Example 117: *Transformer Output Message*

```
<updatedInput>
  <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 381
XSLT Templates	page 383
Common XSLT Functions	page 389

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 118 shows a valid `<xsl:transform>` element for an XSLT script.

Example 118:*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 119 shows a simple XSLT script that transforms `SSN` elements into `acctNum` elements.

Example 119:*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 120, you would specify it as `/name/surname`.

Example 120:*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 120 on page 383 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 121 generates the string `Friday` when run on Example 120 on page 383.

Example 121:*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 122 generates the same output as the script in Example 121.

Example 122:*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 123 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 123: *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 123 as shown in Example 124 using `xsl:apply-templates` and defining a template for the `title` and `subTitle` elements.

Example 124: *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 124:*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 125, 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 125:*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 125: *Bill XML Document*

```
  <amtDue>
    123.50
  </amtDue>
</widgetBill>
```

The XSLT script shown in Example 126 would result in the desired transformation.

Example 126: *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 125 on page 386 with this XSLT script would result in the XML document shown in Example 127.

Example 127:*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 ouput 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 addtion, you can specify a `namespace` for the element using the optional namespace attribute.

SOAP Binding Extensions

This appendix describes the attributes that can be set in the WSDL extensions to configure the Artix SOAP plug-in.

In this appendix

This appendix the following topics:

soap:binding element	page 392
soap:operation element	page 394
soap:body element	page 395
soap:header element	page 398
soap:fault element	page 400
soap:address element	page 402

soap:binding element

Overview

The soap:binding element in a WSDL contract is defined within the binding component, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
```

The soap:binding element is used to signify that SOAP is the message format being used for the binding.

Attributes

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

- style
- transport

style

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

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

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

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

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

If `document` is specified, message parts within the SOAP Body appear directly under the SOAP Body element as body entries and do not appear inside a wrapper element that corresponds to an operation. For example, the SOAP Body of a SOAP request message is as follows if the style is document-based:

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

transport

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

soap:operation element

Overview

A soap:operation element in a WSDL contract is defined within an operation element, which is defined in turn within the binding element, as follows:

```
<binding name="…" type="…" >
    <soap:binding style="…" transport="…">
    <operation name="…" >
        <soap:operation style="…" soapAction="…">
```

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

Attributes

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

- style
- soapAction

style

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

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

See "style" on page 392 for more details of the style attribute.

soapAction

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

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

soap:body element

Overview

A `<soap:body>` element in a binding is a child of the `input`, `output`, and `fault` child elements of the WSDL `operation` element, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
    <operation name="…">
        <soap:operation style="…" soapAction="…">
        <input>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
        </input>
        <output>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
        </output>
        <fault>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
        </fault>
    </operation>
```

A `<soap:body>` element is used to provide information on how message parts are to be appear inside the body of a SOAP message. As explained in "soap:operation element" on page 394, the structure of the SOAP Body within a SOAP message is dependent on the setting of the `soap:operation style` attribute.

Attributes

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

- `use`
- `encodingStyle`
- `namespace`
- `parts`

use

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

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

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

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

- `encoded`
- `literal`

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

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

encodingStyle

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

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

namespace

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

parts

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

soap:header element

Overview

A `soap:header` element in a binding is an optional child of the `input`, `output`, and `fault` elements of the WSDL `operation` element, as follows:

```
<binding name="..." type="...">
    <soap:binding style="..." transport="...">
    <operation name="...">
        <soap:operation style="..." soapAction="...">
        <input>
            <soap:body use="..." encodingStyle="..."
                  namespace="..." parts="..."/>
            <soap:header message="..." part="..." use="..."
                         encodingStyle="..." namespace="..."/>
        </input>
        <output>
            <soap:body use="..." encodingStyle="..."
                  namespace="..." parts="..."/>
            <soap:header message="..." part="..." use="..."
                         encodingStyle="..." namespace="..."/>
        </output>
        <fault>
            <soap:body use="..." encodingStyle="..."
                  namespace="..." parts="..."/>
            <soap:header message="..." part="..." use="..."
                         encodingStyle="..." namespace="..."/>
        </fault>
    </operation>
```

A `soap:header` element defines the information that is placed in a SOAP header element. You can define any number of `soap:header` elements for an operation. As explained in "soap:operation element" on page 394, the structure of the SOAP header within a SOAP message is dependent on the setting of the `soap:operation` element's `style` attribute.

Attributes

Table 28 describes the attributes defined within the `soap:header` element.

Table 28: *Attributes for soap:header*

Configuration Attribute	Explanation
message	Specifies the qualified name of the message from which the contents of the SOAP header is taken.
part	Specifies the name of the message part that is placed into the SOAP header.
use	Used in the same way as the use attribute within the soap:body element. See "use" on page 395 for more details.
encodingStyle	Used in the same way as the encodingStyle attribute within the soap:body element. See "encodingStyle" on page 396 for more details.
namespace	If the soap:operation style attribute is set to rpc, each message part within the SOAP header of a SOAP message is a parameter or return value and will appear inside a wrapper element within the SOAP header. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the soap:header namespace attribute.

soap:fault element

Overview

A `soap:fault` element in a WSDL contract is defined within the `fault` component within an `operation` component, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
    <operation name="…">
        <soap:operation style="…" soapAction="…">
        <input>
            <soap:body use="…" encodingStyle="…">
        </input>
        <output>
            <soap:body use="…" encodingStyle="…">
        </output>
        <fault>
            <soap:fault name="…" use="…" encodingStyle="…"
        </fault>
    </operation>
</binding>
```

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

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

Attributes

Table 29 describes the attributes defined within the `soap:fault` element.

Table 29: *soap:fault attributes*

Configuration Attribute	Explanation
name	This specifies the name of the fault. This relates back to the name attribute for the fault element specified for the corresponding operation within the portType component of the WSDL contract.
use	This attribute is used in the same way as the use attribute within the soap:body element. See "use" on page 395 for more details.
encodingStyle	This attribute is used in the same way as the encodingStyle attribute within the soap:body element. See "encodingStyle" on page 396 for more details.

soap:address element

Overview

The soap:address element in a WSDL contract is defined within the port component within the service component, as follows:

```
<service name="…">
    <port binding="…" name="…">
        <soap:address location="…">
    </port>
</service>
```

The soap:address element is only specified when you want to use SOAP over HTTP.

If you want to use SOAP over IIOP, the element name is iiop:address. Similarly, if you want to use a different payload format over HTTP, the http-conf:client URL attribute is used instead.

Note: When you are using SOAP over HTTP, the http-conf:client and http-conf:server elements can still be validly specified as peer elements of the soap:address element. See "Creating an HTTP Service" on page 255 for more details of http-conf:client and http-conf:server.

Attribute

The soap:address element takes one attribute: location. This specifies the URL of the server to which the client request is being sent.

Valid values are of the form:

```
http://myserver/mypath/
https://myserver/mypath
http://myserver:9001/mypath
http://myserver:9001-9010/mypath
```

CORBA Type Mapping

The Artix CORBA plug-in uses a detailed type map to ensure that data is transmitted without ambiguity.

In this appendix

This appendix discusses the following topics:

Introducing CORBA Type Mapping	page 404
Primitive Type Mapping	page 405
Complex Type Mapping	page 408
Recursive Type Mapping	page 425
Mapping XML Schema Features that are not Native to IDL	page 427
Artix References	page 440

Introducing CORBA Type Mapping

Overview

To ensure that messages are converted into the proper format for a CORBA application to understand, Artix contracts need 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.

Unsupported types

The following CORBA types are not supported:

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

Primitive Type Mapping

Mapping chart

Most primitive IDL types are directly mapped to primitive XML Schema types. Table 30 lists the mappings for the supported IDL primitive types.

Table 30: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type	Artix Java Type
Any	xsd:anyType	corba:any	IT_Bus::AnyHolder	com.iona.webservices.reflect.types.AnyType
boolean	xsd:boolean	corba:boolean	IT_Bus::Boolean	boolean
char	xsd:byte	corba:char	IT_Bus::Char	byte
wchar	xsd:string	corba:wchar		java.lang.String
double	xsd:double	corba:double	IT_Bus::Double	double
float	xsd:float	corba:float	IT_Bus::Float	float
octet	xsd:unsignedByte	corba:octet	IT_Bus::Octet	short
long	xsd:int	corba:long	IT_Bus::Long	int
long long	xsd:long	corba:longlong	IT_Bus::LongLong	long
short	xsd:short	corba:short	IT_Bus::Short	short
string	xsd:string	corba:string	IT_Bus::String	java.lang.String
wstring	xsd:string	corba:wstring		java.lang.String
unsigned short	xsd:unsignedShort	corba:ushort	IT_Bus::UShort	int
unsigned long	xsd:unsignedInt	corba:ulong	IT_Bus::ULong	long
unsigned long long	xsd:unsignedLong	corba:ulonglong	IT_Bus::ULongLong	java.math.BigInteger
TimeBase::UtcT	xsd:dateTime[a]	corba:dateTime	IT_Bus::DateTime	java.util.Calendar

a. The mapping between xsd:dateTime and TimeBase:UtcT is only partial. For the restrictions see "Unsupported time/date values" on page 406

Unsupported types

Artix does not support the CORBA long double type.

Unsupported time/date values

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

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

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

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

Example

The mapping of primitive types is handled in the CORBA binding section of the Artix contract. For example, consider an input message that has a part, score, that is described as an xsd:int as shown in Example 128.

Example 128: *WSDL Operation Definition*

```
<message name="runsScored">
  <part name="score"/>
</message>
<portType ...>
  <operation name="getRuns">
    <input message="tns:runsScored" name="runsScored"/>
  </operation>
</portType>
```

It is described in the CORBA binding as shown in Example 129.

Example 129:*Example CORBA Binding*

```
<binding ...>
  <operation name="getRuns">
    <corba:operation name="getRuns">
      <corba:param name="score" mode="in" idltype="corba:long"/>
    </corba:operation>
    <input/>
    <output/>
  </operation>
</binding>
```

The IDL is shown in Example 130.

Example 130:*getRuns IDL*

```
// IDL
void getRuns(in score);
```

Complex Type Mapping

Overview

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

The `corba:typeMapping` element requires a `targetNamespace` attribute that specifies the namespace for the elements defined by the type map. The default URI is `http://schemas.iona.com/bindings/corba/typemap`. By default, the types defined in the type map are referred to using the `corbatm:` prefix.

Mapping chart

Table 31 shows the mappings from complex IDL types to XML Schema, Artix CORBA type, and Artix C++ types.

Table 31: *Complex Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
`struct`	See Example 132	`corba:struct`	`IT_Bus::SequenceComplexType`
`enum`	See Example 133	`corba:enum`	`IT_Bus::AnySimpleType`
`fixed`	`xsd:decimal`	`corba:fixed`	`IT_Bus::Decimal`
`union`	See Example 138	`corba:union`	`IT_Bus::ChoiceComplexType`
`typedef`	See Example 141		
`array`	See Example 143	`corba:array`	`IT_Bus::ArrayT<>`
`sequence`	See Example 149	`corba:sequence`	`IT_Bus::ArrayT<>`
`exception`	See Example 152	`corba:exception`	`IT_Bus::UserFaultException`

Structures

Mapping

Structures are mapped to `corba:struct` elements. A `corba:struct` element requires three attributes:

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

The elements of the structure are described by a series of `corba:member` elements. The elements must be declared in the same order used in the IDL representation of the CORBA type. A `corba:member` requires two attributes:

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

Example

For example, you may have a structure, `personalInfo`, similar to the one in Example 131.

Example 131:*personalInfo*

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

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

It can be represented in the CORBA type map as shown in Example 132:

Example 132:*CORBA Type Map for personalInfo*

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

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

Enumerations

Mapping

Enumerations are mapped to `corba:enum` elements. A `corba:enum` element requires three attributes:

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

The values for the enumeration are described by a series of `corba:enumerator` elements. The values must be listed in the same order used in the IDL that defines the CORBA enumeration. A `corba:enumerator` element takes one attribute, `value`.

Example

For example, the enumeration defined in Example 131 on page 409, `hairColorType`, can be represented in the CORBA type map as shown in Example 133:

Example 133: *CORBA Type Map for hairColorType*

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

Fixed

Mapping

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

CORBA fixed data types are described using a `corba:fixed` element. A `corba:fixed` element requires five attributes:

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

Example

For example, the fixed type defined in Example 134, `myFixed`, would be

Example 134: *myFixed Fixed Type*

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

described by a type entry in the logical type description of the contract, as shown in Example 135.

Example 135: *Logical description from myFixed*

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

In the CORBA type map portion of the contract, it would be described by an entry similar to Example 136. Notice that the description in the CORBA type map includes the information needed to fully represent the characteristics of this particular fixed data type.

Example 136:*CORBA Type Map for myFixed*

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

Unions

Overview

Unions are particularly difficult to describe using the WSDL framework of an Artix contract. In the logical data type descriptions, the difficulty is how to describe the union without losing the relationship between the members of the union and the discriminator used to select the members. The easiest method is to describe a union using an `xsd:choice` and list the members in the specified order. The OMG's proposed method is to describe the union as an `xsd:sequence` containing one element for the discriminator and an `xsd:choice` to describe the members of the union. However, neither of these methods can accurately describe all the possible permutations of a CORBA union.

Artix Mapping

Artix's IDL compiler generates a contract that describes the logical union using both methods. The description using `xsd:sequence` is named by prepending `_omg_` to the types name. The description using `xsd:choice` is used as the representation of the union throughout the contract.

For example consider the union, `myUnion`, shown in Example 137:

Example 137: *myUnion IDL*

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

This union is described in the logical portion of the contact with entries similar to those shown in Example 138:

Example 138: *myUnion Logical Description*

```
<xsd:complexType name="myUnion">
  <xsd:choice>
    <xsd:element name="case0" type="xsd:string"/>
    <xsd:element name="case12" type="xsd:float"/>
    <xsd:element name="caseDef" type="xsd:int"/>
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="_omg_myUnion4">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1" name="discriminator" type="xsd:short"/>
    <xsd:choice minOccurs="0" maxOccurs="1">
      <xsd:element name="case0" type="xsd:string"/>
      <xsd:element name="case12" type="xsd:float"/>
      <xsd:element name="caseDef" type="xsd:int"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

In the CORBA type map portion of the contract, the relationship between the union's discriminator and its members must be resolved. This is accomplished using a `corba:union` element. A `corba:union` element has four mandatory attributes.

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

The members of the union are described using a series of nested `corba:unionbranch` elements. A `corba:unionbranch` element has two required attributes and one optional attribute.

`name`	A unique identifier used to reference the union member.
`idltype`	The IDL type of the union member. This type can be either a primitive type or another complex type that is defined in the type map.

`default`	The optional attribute specifying if this member is the default case for the union. To specify that the value is the default set this attribute to `true`.

Each `corba:unionbranch` except for one describing the union's default member will have at least one nested `corba:case` element. The `corba:case` element's only attribute, `label`, specifies the value used to select the union member described by the `corba:unionbranch`.

For example `myUnion`, Example 137 on page 414, would be described with a CORBA type map entry similar to that shown in Example 139.

Example 139: *myUnion CORBA type map*

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

Type Renaming

Mapping

Renaming a type using a `typedef` statement is handled using a `corba:alias` element in the CORBA type map. The Artix IDL compiler also adds a logical description for the renamed type in the `types` section of the contract, using an `xsd:simpleType`.

Example

For example, the definition of myLong in Example 140, can be described as

Example 140:*myLong IDL*

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

shown in Example 141:

Example 141:*myLong WSDL*

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

Arrays

Logical mapping

Arrays are described in the logical portion of an Artix contract, using an `xsd:sequence` with its `minOccurs` and `maxOccurs` attributes set to the value of the array's size. For example, consider an array, `myArray`, as defined in Example 142.

Example 142: *myArray IDL*

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

Its logical description will be similar to that shown in Example 143:

Example 143: *myArray logical description*

```
<xsd:complexType name="myArray">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="10" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

In the CORBA type map, arrays are described using a `<corba:array>` element. A `<corba:array>` has five required attributes.

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

For example, the array `myArray` will have a CORBA type map description similar to the one shown in Example 144.

Example 144: *myArray CORBA type map*

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

Multidimensional Arrays

Logical mapping

Multidimensional arrays are handled by creating multiple arrays and combining them to form the multidimensional array. For example, an array defined as shown in Example 145

Example 145: *Multidimensional Array*

```
\\ IDL
typedef long array2d[10][10];
```

generates the logical description shown in Example 146.

Example 146: *Logical Description of a Multidimensional Array*

```
<xsd:complexType name="_1_array2d">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="10" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="array2d">
  <xsd:sequence>
    <xsd:element name="item" type="xsd1:_1_array2d" minOccurs="10" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

The corresponding entry in the CORBA type map is:

Example 147: *CORBA Type Map for a Multidimensional Array*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
  <corba:anonarray name="_2_array2d" type="xsd1:_2_array2d" elemtype="corba:long" bound="10"/>
  <corba:array name="array2d" repositoryID="IDL:array2d:1.0" type="xsd1:array2d"
   elemtype="corbatm:_2_array2d" bound="10"/>
</corba:typeMapping>
```

Sequences

Logical mapping

Because CORBA sequences are an extension of arrays, sequences are described in Artix contracts similarly. Like arrays, sequences are described in the logical type section of the contract using `xsd:sequence` elements. Unlike arrays, the `minOccurs` and `maxOccurs` attributes do not have the same value. `minOccurs` is set to `0` and `maxOccurs` is set to the upper limit of the sequence. If the sequence is unbounded, `maxOccurs` is set to `unbounded`.

For example, the two sequences defined in Example 148, `longSeq` and `charSeq`:

Example 148: *IDL Sequences*

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

are described in the logical section of the contract with entries similar to those shown in Example 149.

Example 149: *Logical Description of Sequences*

```
<xsd:complexType name="longSeq">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="charSeq">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:byte" minOccurs="0" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

In the CORBA type map, sequences are described using a `corba:sequence` element. A `corba:sequence` has five required attributes.

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

`type`	The logical type the structure is mapping.
`elemtype`	The IDL type of the sequence's elements. This type can be either a primitive type or another complex type that is defined within the type map.
`bound`	The size of the sequence.

For example, the sequences described in Example 149 has a CORBA type map description similar to that shown in Example 150:

Example 150:*CORBA type map for Sequences*

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

Exceptions

Mapping

Because exceptions typically return more than one piece of information, they require both an abstract type description and a CORBA type map entry. In the abstract type description, exceptions are described much like structures. In the CORBA type map, exceptions are described using `corba:exception` elements. A `corba:exception` element has three required attributes:

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

The pieces of data returned with the exception are described by a series of `corba:member` elements. The elements must be declared in the same order as in the IDL representation of the exception. A `corba:member` has two required attributes:

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

Example

For example, the exception defined in Example 151, `idNotFound`,

Example 151: *idNotFound Exception*

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

would be described in the logical type section of the contract, with an entry similar to that shown in Example 152:

Example 152:*idNotFound logical structure*

```
<xsd:complexType name="idNotFound">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:short"/>
  </xsd:sequence>
</xsd:complexType>
```

In the CORBA type map portion of the contract, `idNotFound` is described by an entry similar to that shown in Example 153:

Example 153:*CORBA Type Map for idNotFound*

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

Recursive Type Mapping

Overview

Both CORBA IDL and XML Schema allow you define recursive data types. Because both type definition schemes support recursion, Artix directly maps recursive types between IDL and XML Schema. The CORBA typemap generated by Artix to support the CORBA binding is straightforward and directly reflects the recursive nature of the data types.

Defining recursive types in XML Schema

Recursive data types are defined in XML Schema as complex types using the `complexType` element. XML Schema supports two means of defining a recursive type. The first is to have an element of a complex type be of a type that includes an element of the type being defined. Example 154 shows a recursive complex type XML Schema type, `allAboutMe`, defined using a named type.

Example 154: *Recursive XML Schema Type*

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

XML Schema also supports the definition of recursive types using anonymous types. However, Artix does not support this style of defining recursive types.

CORBA typemap

As shown in Example 155, Artix maps recursive types into the CORBA typemap section of the Artix contract as it would non-recursive types, except that it maps the recursive element, which is a sequence in this case, to an

anonymous type using the `corba:anonsequence` element. The `corba:anonsequence` specifies that when IDL is generated from this binding the associated sequence will not generate a new type for itself.

Example 155:*Recursive CORBA Typemap*

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

Generated IDL

While the XML in the CORBA typemap does not explicitly retain the recursive nature of recursive XML Schema types, the IDL generated from the typemap restores the recursion in the IDL type. The IDL generated from the typemap in Example 155 on page 426 defines `allAboutMe` using recursion. Example 156 shows the generated IDL.

Example 156:*IDL for a Recursive Data Type*

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

Mapping XML Schema Features that are not Native to IDL

Overview

There are a number of data types that you can describe in your Artix contract using XML Schema that are not native to IDL. Artix can map these data types into legal IDL so that your CORBA systems can interoperate with applications that use these data type descriptions in their contracts.

These features include:

- Binary Types
- Attributes
- Nested Choices
- Inheritance
- Nillable

Binary Types

Overview

There are three binary types defined in XML Schema that have direct correlation to IDL data-types. These types are:

- `xsd:base64Binary`
- `xsd:hexBinary`
- `soapenc:base64`

These types are all mapped to octet sequences in CORBA.

Example

For example, the schema type, `joeBinary`, described in Example 157 results in the CORBA typemap description shown in Example 158.

Example 157:*joeBinary schema description*

```
<xsd:element name="joeBinary type="xsd:hexBinary"/>
```

The resulting IDL for `joeBinary` is shown in Example 159.

Example 158:*joeBinary CORBA typemap*

```
<corba:sequence name="joeBinary" bound="0"
 elemtype="corba:octet" repositoryID="IDL:joeBinary:1.0"
 type="xsd:hexBinary"/>
```

The mappings for `xsd:base64Binary` and `soapenc:base64` would be similar except that the `type` attribute in the CORBA typemap would specify the appropriate type.

Example 159:*joeBinary IDL*

```
\\IDL
typedef sequence<octet> joeBinary;
```

Attributes

Mapping

Required XML Schema attributes are treated as normal elements in a CORBA structure.

Note: Attributes are not supported for complex types defined with `choice` element.

Example

For example, the complex type, `madAttr`, described in Example 160 contains two attributes, `material` and `size`.

Example 160: *madAttr XML Schema*

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

`madAttr` would generate the CORBA typemap shown in Example 161. Notice that `size` and `material` are simply incorporated into the `madAttr` structure in the CORBA typemap.

Example 161: *madAttr CORBA typemap*

```
<corba:annonstring bound="3" name="materialType" type="tns:material"/>
<corba:struct name="madAttr" repositoryID="IDL:madAttr:1.0" type="typens:madAttr">
  <corba:member name="style" idltype="corba:string"/>
  <corba:member name="gender" idltype="corba:char"/>
  <corba:member name="size" idltype="corba:long"/>
  <corba:member name="material" idltype="ns1:materialType"/>
</corba:struct>
```

Similarly, in the IDL generated using a contract containing `madAttr`, the attributes are made elements of the structure and are placed in the order in which they are listed in the contract. The resulting IDL structure is shown in Example 162.

Example 162:*madAttr IDL*

```
\\IDL
struct madAttr
{
  string style;
  char gender;
  long size;
  string<3> material;
}
```

Nested Choices

Mapping

When mapping complex types containing nested `xsd:choice` elements into CORBA, Artix will break the nested `xsd:choice` elements into separate `unions` in CORBA. The resulting union will have the name of the original complex type with `ChoiceType` appended to it. So, if the original complex type was named *`joe`*, the union representing the nested choice would be named `joeChoiceType`.

The nested choice in the original complex type will be replaced by an element of the new union created to represent the nested choice. This element will have the name of the new union with `_f` appended. So if the original structure was named `carla`, the replacement element will be named `carlaChoiceType_f`.

The original type description will not be changed, the break out will only appear in the CORBA typemap and in the resulting IDL.

Example

For example, the complex type `details`, shown in Example 163, contains a nested `choice`.

Example 163:*details XML Schema*

```
<complexType name="Details">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="address" type="xsd:string"/>
    <choice>
      <element name="employer" type="xsd:string"/>
      <element name="unemploymentNumber" type="xsd:int"/>
    </choice>
  </sequence>
</complexType>
```

The resulting CORBA typemap, shown in Example 164, contains a new union, `detailsChoiceType`, to describe the nested choice. Note that the `type` attribute for both `details` and `detailsChoiceType` has the name of the

original complex type defined in the schema. The nested choice is represented in the original structure as a member of type `detailsChoiceType`.

Example 164:*details CORBA typemap*

```
<corba:struct name="details" repositoryID="IDL:details:1.0" type="xsd1:details">
  <corba:member idltype="corba:string" name="name"/>
  <corba:member idltype="corba:string" name="address"/>
  <corba:member idltype="ns1:detailsChoiceType" name="detailsChoiceType_f"/>
</corba:struct>
<corba:union discriminator="corba:long" name="detailsChoiceType"
             repositoryID="IDL:detailsChoiceType:1.0" type="xsd1:details">
  <corba:unionbranch idltype="corba:string" name="employer">
    <corba:case label="0"/>
  </corba:unionbranch>
  <corba:unionbranch idltype="corba:long" name="unemploymentNumber">
    <corba:case label="1"/>
  </corba:unionbranch>
</corba:union>
```

The resulting IDL is shown in Example 165.

Example 165:*details IDL*

```
\\IDL
union detailsChoiceType switch(long)
{
  case 0:
    string employer;
  case 1:
    long unemploymentNumber;
};
struct details
{
  string name;
  string address;
  detailsChoiceType DetailsChoiceType_f;
};
```

Inheritance

Mapping

XML Schema describes inheritance using the `complexContent` tag and the `extension` tag. For example the complex type `seaKayak`, described in Example 166, extends the complex type `kayak` by including two new fields.

Example 166: *seaKayak XML Schema*

```
<complexType name="kayak">
  <sequence>
    <element name="length" type="xsd:int"/>
    <element name="width" type="xsd:int"/>
    <element name="material" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="seaKayak">
  <complexContent>
    <extension base="kayak">
      <sequence>
        <element name="chines" type="xsd:string"/>
        <element name="cockpitStyle" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

When complex types using `complexContent` are mapped into CORBA types, Artix creates generates an intermediate type to represent the complex data defined within the `complexContent` element. The intermediate type is named by appending an identifier describing the complex content to the new type's name. Table 32 shows the complex content identifiers used appended to the intermediate type name.

Table 32: *Complex Content Identifiers in CORBA Typemap*

XML Schema Type	Typemap Identifier
`sequence`	`SequenceStruct`
`all`	`AllStruct`
`choice`	`ChoiceType`

The CORBA type generated to represent the XML Schema type generated to represent the type derived by extension will have an element of the type that it extends, named *baseType_f* and an element of the intermediate type, named *intermediateType_f*. Any attributes that are defined in the extended type are then mapped into the new CORBA type following the rules for mapping XML Schema attributes into CORBA types.

Example

Example 167 shows how Artix maps the complex types defined in Example 166 on page 433 into a CORBA type map.

Example 167: *seaKayak CORBA type map*

```
<corba:struct name="kayak" repositoryID="IDL:kayak:1.0" type="tns:kayak">
  <corba:element name="length" idltype="corba:long"/>
  <corba:element name="width" idltype="corba:long"/>
  <corba:element name="material" idltype="corba:string"/>
</corba:struct>
<corba:struct name="seaKayak" repositoryID="IDL:seaKayak:1.0" type="tns:seaKayak">
  <corba:element name="kayak_f" idltype="ns1:kayak"/>
  <corba:element name="seaKayakSequenceStruct_f" idltype="ns1:seaKayakSequenceStruct"/>
</corba:struct>
<corba:struct name="seaKayakSequenceStruct" repositoryID="IDL:seaKayakSequenceStruct:1.0"
              type="tns:seaKayakSequenceStruct">
  <corba:element name="chines" idltype="corba:string"/>
  <corba:element name="cockpitStyle" idltype="corba:string"/>
</corba:struct>
```

The IDL generated by Artix for the types defined in Example 166 on page 433 is shown in Example 168.

Example 168: *seaKayak IDL*

```
\\ IDL
struct kayak
{
  long length;
  long width;
  string material;
};
struct seaKayakSequenceStruct
{
  string chines;
  string cockpitStyle;
};
```

Example 168:*seaKayak IDL*

```
struct seaKayak
{
  kayak kayak_f;
  seaKayakSequenceStruct seqKayakSequenceStruct_f;
};
```

Nillable

Mapping

XML Schema supports an optional attribute, `nillable`, that specifies that an element can be `nil`. Setting an element to `nil` is different than omitting an element whose `minOccurs` attribute is set to `0`; the element must be included as part of the data sent in the message.

Elements that have `nillable="true"` set in their logical description are mapped to a CORBA union with a single case, `TRUE`, that holds the value of the element if it is not set to `nil`.

Example

For example, imagine a service that maintains a database of information on people who download software from a web site. The only required piece of information the visitor needs to supply is their zip code. Optionally, visitors can supply their name and e-mail address. The data is stored in a data structure, `webData`, shown in Example 169.

Example 169: *webData XML Schema*

```
<complexType name="webData">
  <sequence>
    <element name="zipCode" type="xsd:int"/>
    <element name="name" type="xsd:string" nillable="true/>
    <element name="emailAddress" type="xsd:string"
             nillable="true"/>
  </sequence>
</complexType>
```

When webData is mapped to a CORBA binding, it will generate a union, string_nil, to provide for the mapping of the two nillable elements, name and emailAddress. Example 170 shows the CORBA typemap for webData.

Example 170:*webData CORBA Typemap*

```
<corba:typemapping ...>
  <corba:struct name="webData" repositoryID="IDL:webData:1.0" type="xsd1:webData">
    <corba:member idltype="corba:long" name="zipCode"/>
    <corba:member idltype="ns1:string_nil" name="name"/>
    <corba:member idltype="ns1:string_nil" name="emailAddress"/>
  </corba:struct>
  <corba:union discriminator="corba:boolean" name="string_nil" repositoryID="IDL:string_nil:1.0"
               type="xsd1:emailAddress">
    <corba:unionbranch idltype="corba:string" name="value">
      <corba:case label="TRUE"/>
    </corba:unionbranch>
  </corba:union>
</corba:typeMapping>
```

The type assigned to the union, string_nil, does not matter as long as the type assigned maps back to an xsd:string. This is true for all nillable element types.

Example 171 shows the IDL for webData.

Example 171:*webData IDL*

```
\\IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
struct webData {
    long zipCode;
    string_nil name;
    string_nil emailAddress;
};
```

Optional Attributes

Overview

Attributes defined as optional in XML Schema are mapped similar to `nillable` elements. Attributes that do not have `use="required"` set in their logical description are mapped to a CORBA union with a single case, `TRUE`, that holds the value of the element if it is set.

> **Note:** By default attributes are optional if `use` is not set to `required`.

For example, you could define the complex type in Example 169 using attributes instead of a sequence. The data description for `webData` defined with attributes is shown in Example 172.

Example 172: *webData XML Schema Using Attributes*

```
<complexType name="webData">
  <attribute name="zipCode" type="xsd:int" use="required"/>
  <attribute name="name" type="xsd:string"/>
  <attribute name="emailAddress" type="xsd:string"/>
</complexType>
```

CORBA type mapping

When `webData` is mapped to a CORBA binding, it will generate a union, `string_nil`, to provide for the mapping of the two nillable elements, name and `emailAddress`. Example 173 shows the CORBA typemap for `webData`.

Example 173: *webData CORBA Typemap*

```
<corba:typemapping ...>
  <corba:union discriminator="corba:boolean" name="string_nil" repositoryID="IDL:string_nil:1.0"
               type="xsd1:emailAddress">
    <corba:unionbranch idltype="corba:string" name="value">
      <corba:case label="TRUE"/>
    </corba:unionbranch>
  </corba:union>
  <corba:struct name="webData" repositoryID="IDL:webData:1.0" type="xsd1:webData">
    <corba:member idltype="corba:long" name="zipCode"/>
    <corba:member idltype="ns1:string_nil" name="name"/>
    <corba:member idltype="ns1:string_nil" name="emailAddress"/>
  </corba:struct>
</corba:typeMapping>
```

The type assigned to the union, string_nil, does not matter as long as the type assigned maps back to an xsd:string. This is true for all optional attributes.

Example 174 shows the IDL for webData.

Example 174:*webData IDL*

```
\\IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
struct webData {
    long zipCode;
    string_nil name;
    string_nil emailAddress;
};
```

Artix References

Overview

Artix references provide a means of passing a reference to a service between two operations. Because Artix services are Web services, their references are very different than references used in CORBA. Artix does, however, provide a mechanism for passing Artix references to CORBA applications over the Artix CORBA transport. This functionality allows CORBA applications to make calls on Artix services that return references to other Artix services.

For a detailed discussion of Artix references read Developing Artix Applications in C++.

Specifying references to map to CORBA

Artix references are mapped into a CORBA in one of two ways. The simplest way is to just specify your reference types as you would for an Artix service using SOAP. In this case, the Artix references are mapped into generic CORBA `Objects`.

The second method allows you to generate type-specific CORBA references, but requires some planning in the creation of your XML Schema type definitions. When creating a reference type, you can specify the name of the CORBA binding that describes the interface in the physical section of the contract using an `xsd:annotation` element. Example 175 shows the syntax for specifying the binding in the type definition.

Example 175: *Reference Binding Specification*

```
<xsd:element name="typeName" type="references:Reference">
  <xsd:annotation>
    <xsd:appinfo>corba:binding=CORBABindingName</xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

When you specify a reference using the annotation, the CORBA binding generator and the IDL generator will inspect the specified binding and create a type-specific reference in the IDL generated for the contract that allows you to make use of the reference.

> **Note:** Before you can generate a type-specific reference you need to generate the CORBA binding of the referenced interface.

CORBA typemap representation

Artix references are mapped to `corba:object` elements in the CORBA typemap section of an Artix contract. `corba:object` elements have four attributes:

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

Example

Example 176 shows an Artix contract fragment that uses Artix references.

Example 176:*Reference Sample*

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

Example 176: *Reference Sample (Continued)*

```
  <types>
    <schema
     targetNamespace="http://schemas.myBank.com/bankTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:import schemaLocation="./references.xsd"
                 namespace="http://schemas.iona.com/references"/>
...
      <xsd:element name="account" type="references:Reference">
        <xsd:annotation>
          <xsd:appinfo>
          corba:binding=AccountCORBABinding
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:element>
  </schema>
</types>
...
  <message name="find_accountResponse">
    <part name="return" element="xsd1:account"/>
  </message>
  <message name="create_accountResponse">
    <part name="return" element="xsd1:account"/>
  </message>
  <portType name="Account">
    <operation name="account_id">
      <input message="tns:account_id" name="account_id"/>
      <output message="tns:account_idResponse"
              name="account_idResponse"/>
    </operation>
    <operation name="balance">
      <input message="tns:balance" name="balance"/>
      <output message="tns:balanceResponse"
              name="balanceResponse"/>
    </operation>
    <operation name="withdraw">
      <input message="tns:withdraw" name="withdraw"/>
      <output message="tns:withdrawResponse"
              name="withdrawResponse"/>
      <fault message="tns:InsufficientFundsException"
   name="InsufficientFunds"/>
    </operation>
```

Example 176:*Reference Sample (Continued)*

```
    <operation name="deposit">
      <input message="tns:deposit" name="deposit"/>
      <output message="tns:depositResponse"
              name="depositResponse"/>
    </operation>
  </portType>
  <portType name="Bank">
    <operation name="find_account">
      <input message="tns:find_account" name="find_account"/>
      <output message="tns:find_accountResponse"
              name="find_accountResponse"/>
      <fault message="tns:AccountNotFound"
             name="AccountNotFound"/>
    </operation>
    <operation name="create_account">
      <input message="tns:create_account" name="create_account"/>
      <output message="tns:create_accountResponse"
              name="create_accountResponse"/>
      <fault message="tns:AccountAlreadyExistsException"
             name="AccountAlreadyExists"/>
    </operation>
  </portType>
</definitions>
```

The element named `account` is a reference to the interface defined by the `Account` port type and the `find_account` operation of `Bank` returns an element of type `account`. The annotation element in the definition of `account` specifies the binding, `AccountCORBABinding`, of the interface to which the reference refers. Because you typically create the data types before you create the bindings, you must be sure that the generated binding name matches the name you specified. This can be controlled using the `-b` flag to `wsdltocorba`.

The first step to generating the `Bank` interface to use a type-specific reference to an `Account` is to generate the CORBA binding for the `Account` interface. You would do this by using the command **`wsdltocorba -corba -i Account -b AccountCORBABinding`** *`wsdlName`***`.wsdl`** and replace *`wsdlName`* with the name of your contract. Once you have generated the CORBA binding for the `Account` interface, you can generate the CORBA binding and IDL for the `Bank` interface.

[Example 177](#) shows the generated CORBA typemap resulting from generating both the `Account` and the `Bank` interfaces into the same contract.

Example 177:*CORBA Typemap with References*

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

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

[Example 178](#) shows the IDL generated for the Bank interface.

Example 178:*IDL Generated From Artix References*

```
//IDL
...
interface Account
{
  string account_id();
  float balance();
  void withdraw(in float amount)
    raises(::InsufficientFundsException);
  void deposit(in float amount);
};
interface Bank
{
  ::Account find_account(in string account_id)
    raises(::AccountNotFoundException);
  ::Account create_account(in string account_id,
                           in float initial_balance)
    raises(::AccountAlreadyExistsException);
};
```

TibrvMsg Default Mappings

The default mappings between the logically defined messages in an Artix contract and a TibrvMsg are sufficient for most cases.

TIBRVMSG type mapping

Table 33 shows how TibrvMsg data types are mapped to XSD types in Artix contracts.

Table 33: *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

Table 33: *TIBCO to XSD Type Mapping (Continued)*

TIBRVMSG	XSD
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
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

Table 33: *TIBCO to XSD Type Mapping (Continued)*

TIBRVMSG	XSD
TIBRVMSG_STRING	xsd:Name
TIBRVMSG_STRING	xsd:NCName
TIBRVMSG_STRING	xsd:ID

a. While TIBRVMSG_DATATIME 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 sequence's elements are the 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 field's 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 all's elements are the 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

Built-in schema type NMTOKENs 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 instances 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 TibrvMsg 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.

APPENDIX D

HTTP Port Properties

HTTP has a number of settings that can be specified in the transport definition.

In this appendix

This appendix contains the following topics:

Defining an HTTP Port	page 452
HTTP Client Configuration	page 453
HTTP Server Configuration	page 456
HTTP Attribute Details	page 459

Defining an HTTP Port

Overview

To allow you more flexibility in configuring an HTTP port, Artix has its own set of WSDL extensions that can be used to define an HTTP port. All of the configuration elements are optional.

An HTTP port is fully defined by the address element.

Using the HTTP extension

Example 179 shows the namespace entries you need to add to the `definitions` element of your contract to use the HTTP extensions.

Example 179: *Artix HTTP Extension Namespaces*

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

HTTP configuration elements

Because HTTP client ports and HTTP server ports have slightly different configuration options, Artix uses two elements to configure an HTTP port:

- `http-conf:client` defines a client port.
- `http-conf:server` defines a server port.

HTTP Client Configuration

Confirguration attributes

Table 34 describes the client-side configuration attributes for the HTTP transport that are defined within the `http-conf:client` element.

In most cases, a link is provided to additional detail on the attribute in the "HTTP Attribute Details" section.

Table 34: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`SendTimeout`	Specifies the length of time the client tries to send a request to the server before the connection is timed out.
`ReceiveTimeout`	Specifies the length of time the client tries to receive a response from the server before the connection is timed out.
`AutoRedirect`	Specifies if a request should be automatically redirected when the server issues a redirection reply via `RedirectURL`.
`UserName`	Specifies the user name that is to be used for authentication.
`Password`	Specifies the password that is to be used for authentication.
`AuthorizationType`	Specifies the name of the authorization scheme in use.
`Authorization`	Specifies the authorization credentials used to perform the authorization.
`Accept`	Specifies what media types the client is prepared to handle.
`AcceptLanguage`	Specifies the client's preferred language for receiving responses.

Table 34: *HTTP Client Configuration Attributes (Continued)*

Configuration Attribute	Explanation
AcceptEncoding	Specifies what content codings the client is prepared to handle.
ContentType	Specifies the media type of the data being sent in the body of the client request.
Host	Specifies the internet host and port number of the resource on which the client request is being invoked.
Connection	Specifies whether a particular connection is to be kept open or closed after each request/response dialog.
ConnectionAttempts	Specifies the number of times a client will transparently attempt to connect to server.
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
Cookie	Specifies a static cookie to be sent to the server.
BrowserType	Specifies information about the browser from which the client request originates.
Referer	Specifies the URL of the resource that provided the hyperlink to the requested service.
ProxyServer	Specifies the URL of the proxy server, if one exists along the message path.
ProxyUserName	Specifies the username to use for authentication on the proxy server if it requires separate authorization.

Table 34: *HTTP Client Configuration Attributes (Continued)*

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

HTTP Server Configuration

Configuration attributes

Table 35 describes the server-side configuration attributes for the HTTP transport that are defined within the `http-conf:server` element.

In most cases, a link is provided to additional detail on the attribute in the "HTTP Attribute Details" section

Table 35: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`SendTimeout`	Sets the length of time the server tries to send a response to the client before the connection times out.
`ReceiveTimeout`	Sets the length of time the server tries to receive a client request before the connection times out.
`SuppressClientSendErrors`	Specifies whether exceptions are to be thrown when an error is encountered on receiving a client request.
`SuppressClientReceiveErrors`	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a client.
`HonorKeepAlive`	Specifies whether the server should honor client requests for a connection to remain open after a server response has been sent to a client.
`RedirectURL`	Sets the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource.

Table 35: *HTTP Server Configuration Attributes (Continued)*

Configuration Attribute	Explanation
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a server to a client.
ContentLocation	Sets the URL where the resource being sent in a server response is located.
ContentType	Sets the media type of the information being sent in a server response, for example, text/html or image/gif.
ContentEncoding	Specifies what additional content codings have been applied to the information being sent by the server.
ServerType	Specifies what type of server is sending the response to the client.
UseSecureSockets	Indicates whether the server wants a secure HTTP connection running over SSL or TLS.
ServerCertificate	Sets the full path to the PKCS12-encoded X509 certificate issued by the certificate authority for the server.
ServerCertificateChain	Sets the full path to the file that contains all the certificates in the server's certificate chain.
ServerPrivateKey	Sets the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by ServerCertificate.

Table 35: *HTTP Server Configuration Attributes (Continued)*

Configuration Attribute	Explanation
`ServerPrivateKeyPassword`	Sets a password that is used to decrypt the PKCS12-encoded private key, if it has been encrypted with a password.
`TrustedRootCertificate`	Sets the full path to the PKCS12-encoded X509 certificate for the certificate authority. This is used to validate the certificate presented by the client.

HTTP Attribute Details

Overview

This section provides more detail on the client and server attributes listed in Table 34 and Table 35.

SendTimeout

Overview

`SendTimeout` specifies the length of time, in milliseconds, that an application can continue to try to send a message before the connection times out. The default is `30000`.

This property is valid for both servers and clients.

Example

Example 180 shows a definition of an HTTP client that will attempt to send a request for one minute before the connection times out.

Example 180: *HTTP SendTimeout Example*

```
<http-conf:client SendTimeout="60000"/>
```

ReceiveTimeout

Overview

`ReceiveTimeout` specifies the length of time, in milliseconds, that an application can continue to try to recieve a message before the connection times out. The default is `30000`.

This property is valid for both servers and clients.

Example

Example 181 shows a definition of an HTTP server that will attempt to receive a request for one minute before the connection times out.

Example 181: *HTTP ReceiveTimeout Example*

```
<http-conf:server ReceiveTimeout="60000"/>
```

AutoRedirect

Overview

AutoRedirect is a client-side property that specifies whether a client request should be automatically redirected on behalf of the client when the server issues a redirection reply via the RedirectURL server-side configuration attribute. Valid values are true and false. The default is false, to let the client redirect the request itself.

UserName

Overview

To ensure a level of security, some servers require that client users be authenticated before their requests are processed. In the case of basic authentication, the server requires the client user to supply a username and password. `UserName` specifies the user name that is to be used for authentication. If set, `UserName` is sent as a transport attribute in the header of request messages from the client to the server.

Password

Overview

To ensure a level of security, some servers require that client users be authenticated before their requests are processed. In the case of basic authentication, the server requires the client user to supply a username and password. `Password` specifies the password that is to be used for authentication. If set, `Password` is sent as a transport attribute in the header of request messages from the client to the server.

AuthorizationType

Overview

`AuthorizationType` specifies the name of the authorization scheme in use. This information is specified and handled at application level. Artix does not perform any validation on this value. It is the user's responsibility to ensure that the correct scheme name is specified, as appropriate. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

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

Authorization

Overview

`Authorization` specifies the authorization credentials used to perform the authorization. These are encoded and handled at application-level. Artix does not perform any validation on the specified value. It is the user's responsibility to ensure that the correct authorization credentials are specified, as appropriate. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

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

Accept

Overview

`Accept` specifies what media types the client is prepared to handle. These are also known as multipurpose internet mail extensions (MIME) types. MIME types are regulated by the Internet Assigned Numbers Authority (IANA). See http://www.iana.org/assignments/media-types/ for more details.

If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Specifying the media type

Specified values consist of a main type and sub-type, separated by a forward slash. For example, a main type of `text` might be qualified as follows: `text/html` or `text/xml`. Similarly, a main type of image might be qualified as follows: `image/gif` or `image/jpeg`.

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

Examples of typical types that might be set are:

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

A full list of MIME types is available at http://www.iana.org/assignments/media-types/.

AcceptLanguage

Overview

`AcceptLanguage` specifies what language (for example, American English) the client prefers for the purposes of receiving a response. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Specifying the language

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

More details

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

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

AcceptEncoding

Overview

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

If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Content encodings

Content codings are regulated by the Internet Assigned Numbers Authority (IANA). Possible content coding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html for more details of content codings.

ContentType

Overview

`ContentType` specifies the media type of the data being sent in the body of a message. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server. Flor clients this is only relevant if the client request specifies the `POST` method to send data to the server for processing.

Type specification

Types are specified as multipurpose internet mail extensions (MIME) types. MIME types are regulated by the Internet Assigned Numbers Authority (IANA). See http://www.iana.org/assignments/media-types/ for more details.

Specified values consist of a main type and sub-type, separated by a forward slash. For example, a main type of `text` might be qualified as follows: `text/html` or `text/xml`. Similarly, a main type of image might be qualified as follows: `image/gif` or `image/jpeg`.

The default type is `text/xml`. Other specifically supported types include: `application/jpeg`, `application/msword`, `application/xbitmap`, `audio/au`, `audio/wav`, `text/html`, `text/text`, `image/gif`, `image/jpeg`, `video/avi`, `video/mpeg`. Any content that does not fit into any type in the preceding list should be specified as `application/octet-stream`.

Typical client settings

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

ContentEncoding

Overview

`ContentEncoding` can be used in conjunction with `ContentType`. It 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.

The primary use of `ContentEncoding` is to allow a document to be compressed using some encoding mechanism, such as zip or gzip.

If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.

Host

Overview

`Host` specifies the internet host and port number of the resource on which the client request is being invoked. This is sent by default based upon the URL specified in the `URL` attribute. It indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same internet protocol (IP) address).

Note: Certain DNS scenarios or application designs might request you to set this, but it is not typically required.

If set, `Host` is sent as a transport attribute in the header of a request message from the client to the server.

Connection

Overview

`Connection` specifies whether a particular connection is to be kept open or closed after each request/response dialog. Valid values are `close` and `Keep-Alive`. The default is `Keep-Alive`, specifying that the client wishes to maintain the connection to the server after the initial request is processed. If the server honors the request, the connection is reused for subsequent request/response dialogs. If `close` is specified, the connection to the server is closed after each request/response dialog.

Note: The server can choose to not honor a request to keep the connection open, and many servers and proxies (caches) do not honor such requests.

CacheControl

Overview

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

Client-side

Table 36 shows the valid settings for CacheControl in http-conf:client.

Table 36: *Settings for http-conf:client CacheControl*

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent client requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of a response or any part of the request that invoked it.
max-age	The client can accept a response whose age is no greater than the specified time in seconds.
max-stale	The client can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the client can still accept that response. If no value is assigned, it means the client can accept a stale response of any age.
min-fresh	The client wants a response that will be still be fresh for at least the specified number of seconds indicated.

Table 36: *Settings for http-conf:client CacheControl (Continued)*

Directive	Behavior
`no-transform`	Caches must not modify media type or location of the content in a response between a server and a client.
`only-if-cached`	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
`cache-extension`	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

Server-side

Table 37 shows the valid values for `CacheControl` in `http-conf:server`.

Table 37: *Settings for http-conf:server CacheControl*

Directive	Behavior
`no-cache`	Caches cannot use a particular response to satisfy subsequent client requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`public`	Any cache can store the response.

Table 37: *Settings for http-conf:server CacheControl (Continued)*

Directive	Behavior
`private`	Public (*shared*) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
`no-store`	Caches must not store any part of response or any part of the request that invoked it.
`no-transform`	Caches must not modify the media type or location of the content in a response between a server and a client.
`must-revalidate`	Caches must revaildate expired entries that relate to a response before that entry can be used in a subsequent response.
`proxy-revelidate`	Means the same as `must-revalidate`, except that it can only be enforced on shared caches and is ignored by private unshared caches. If using this directive, the `public` cache directive must also be used.
`max-age`	Clients can accept a response whose age is no greater that the specified number of seconds.
`s-maxage`	Means the same as `max-age`, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by `s-maxage` overrides the age specified by `max-age`. If using this directive, the `proxy-revalidate` directive must also be used.

Table 37: *Settings for http-conf:server CacheControl (Continued)*

Directive	Behavior
`cache-extension`	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

Cookie

Overview

`Cookie` specifies a static cookie to be sent to the server. Some session designs that maintain state use cookies to identify sessions.

> **Note:** If the cookie is dynamic, it must be set by the server when the server is first accessed, and is then handled automatically by the application runtime.

BrowserType

Overview

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

Referer

Overview

If a client request is as a result of the browser user clicking on a hyperlink rather than typing a URL, this specifies the URL of the resource that provided the hyperlink.

This is sent automatically if `AutoRedirect` is set to `true`. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications.

If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

ProxyServer

Overview

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

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

ProxyAuthorizationType

Overview

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

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

ProxyAuthorization

Overview

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

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

UseSecureSockets

Overview

`UseSecureSockets` indicates if the application wants to open a secure connection using SSL or TLS. A secure HTTP connection is commonly referred to as HTTPS.

Valid values are `true` and `false`. The default is `false`, to indicate that the client does not want to open a secure connection.

> **Note:** If the `http-conf:client URL` attribute has a value with a prefix of `https://`, a secure HTTP connection is automatically enabled, even if `UseSecureSockets` is not set to `true`.

ClientPrivateKey

Overview

`ClientPrivateKey` is used in conjunction with `ClientCertificate`. It specifies the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by `ClientCertificate`.

This is required only if `ClientCertificate` has been specified.

SuppressClientSendErrors

Overview

`SuppressClientSendErrors` specifies whether exceptions are to be thrown when an error is encountered on receiving a client request.

Valid values are `true` and `false`. The default is `false`, to throw exceptions on encountering errors.

SuppressClientReceiveErrors

Overview

`SuppressClientReceiveErrors` specifies whether exceptions are to be thrown when an error is encountered on sending a response to a client.

Valid values are `true` and `false`. The default is `false`, to throw exceptions on encountering errors.

HonorKeepAlive

Overview

`HonorKeepAlive` specifies whether the server should honor client requests for a connection to remain open after a server response has been sent to a client. Servers can achieve higher concurrency per thread by honoring requests to keep connections alive.

Values

HonorKeepAlive can be set to `true` or `false`:

- `true`(default) specifies that the request socket is kept open provided the client is using at least version 1.1 of HTTP and has requested that the connection is kept alive. Otherwise, the connection is closed.
- `false` specifies that the socket is automatically closed after a server response is sent.

RedirectURL

Overview

`RedirectURL` specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource.

In this case, if a status code is not automatically set in the first line of the server response, the status code is set to `302` and the status description is set to `Object Moved`.

If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.

ServerType

Overview

`ServerType` Specifies what type of server is sending the response to the client. If set, it is sent as a transport attribute in the header of a response message from the server to the client.

Specifying the value

Values in this case take the form *`program-name/version`*. For example, `Apache/1.2.5`.

ServerCertificateChain

Overview

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

WebSphere MQ Port Properties

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

Overview

To enable Artix to interoperate with WebSphere MQ, you must describe the WebSphere MQ port in the Artix contract defining the behavior of your Artix instance. Artix uses a number of WSDL extensions to specify all of the attributes that can be set on an WebSphere MQ port. The XML Schema describing the extensions used for the WebSphere MQ port definition is included in the Artix installation under the `schemas` directory.

Defining an MQ Port

MQ port attributes

Table 38 lists the attributes that are use to define the properties of a WebSphere MQ port.

Each attribute is described in detail in the sections that follow the table.

Table 38: *WebSphere MQ Port Attributes*

Attributes	Description
QueueManager	Specifies the name of the queue manager.
QueueName	Specifies the name of the message queue.
ReplyQueueName	Specifies the name of the queue where response messages are received. This setting is ignored by WebSphere MQ servers when the client specifies the ReplyToQ in the request message's message descriptor.
ReplyQueueManager	Specifies the name of the reply queue manager. This setting is ignored by WebSphere MQ servers when the client specifies the ReplyToQMgr in the request message's message descriptor.
Server_Client	Specifies that the client libraries are to be used because the application is using queues on a remote machine.
ModelQueueName	Specifies the name of the queue to be used as a model for creating dynamic queues.
AliasQueueName	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.
ConnectionName	Specifies the name of the connection by which the adapter connects to the queue.

Table 38: *WebSphere MQ Port Attributes (Continued)*

Attributes	Description
ConnectionReusable	Specifies if the connection can be used by more than one application.
ConnectionFastPath	Specifies if the queue manager will be loaded in process.
UsageStyle	Specifies if messages can be queued without expecting a response.
CorrelationStyle	Specifies what identifier is used to correlate request and response messages.
AccessMode	Specifies the level of access applications have to the queue.
Timeout	Specifies the amount of time within which the send and receive processing must begin before an error is generated.
MessageExpiry	Specifies the value of the MQ message descriptor's Expiry field.
MessagePriority	Specifies the value of the MQ message descriptor's Priority field.
Delivery	Specifies the value of the MQ message descriptor's Persistence field.
Transactional	Specifies if transaction operations must be performed on the messages.
ReportOption	Specifies the value of the MQ message descriptor's Report field.
Format	Specifies the value of the MQ message descriptor's Format field.
MessageId	Specifies the value for the MQ message descriptor's MsgId field.

Table 38: *WebSphere MQ Port Attributes (Continued)*

Attributes	Description
CorrelationId	Specifies the value for the MQ message descriptor's CorrelId field.
ApplicationData	Specifies optional information to be associated with the message.
AccountingToken	Specifies the value for the MQ message decscriptor's AccountingToken field.
Convert	Specifies if the messages in the queue need to be converted to the system's native encoding.
ApplicationIdData	Specifies the value for the MQ message descriptor's ApplIdentityData field.
ApplicationOriginData	Specifies the value for the MQ message descriptor's ApplOriginData field.
UserIdentification	Specifies the value for the MQ message descriptor's UserIdentifier field.

MQ Port Attributes

Overview

This section provides more detail on the attributes listed in WebSphere MQ Port Attributes page 494.

QueueManager

Overview

`QueueManager` specifies the name of the WebSphere MQ queue manager used for request messages. Client applications will use this queue manager to place requests and server applications will use this queue manager to listen for request messages. You must provide this information when configuring a WebSphere MQ port.

Example

Example 182 shows a simple WebSphere MQ server port configuration for servers that listen for requests using a queue manager called `leo`.

Example 182: *MQ Port Definition*

```
<mq:server QueueManager="leo" QueueName="requestQ"/>
```

QueueName

Overview

`QueueName` is a required attribute for a WebSphere MQ port. It specifies the request message queue. Client applications place request messages into this queue. Server applications take requests from this queue. The queue must be configured under the specified queue manager before it can be used.

Example

Example 183 shows a definition of a simple WebSphere MQ client that places oneway requests onto a queue called `ether`.

Example 183: *WebSphere MQ QueueName example*

```
<mq:client QueueManager="Qmgr" QueueName="ether"/>
```

ReplyQueueName

Overview

ReplyQueueName is mapped to the MQ message descriptor's ReplyToQ field. It specifies the name of the reply message queue used by the port. When configuring an MQ client port this attribute is required if the clients expect replies to their requests. When configuring an MQ server port you can leave this attribute unset if you are sure that all clients are populating the ReplyToQ field in the message descriptor of their requests.

Server handling of ReplyQueueName

When a WebSphere MQ server receives a request, it first looks at the request's message descriptor's ReplyToQ field. If the request's message descriptor has ReplyToQ set, the server uses the reply queue specified in the message descriptor and ignores the ReplyQueueName setting. If the ReplyToQ field in the message descriptor is not set, the server will use the ReplyQueueName to determine where to send reply messages.

Example

Example 184 shows a WebSphere MQ server port that defaults to placing reply messages onto the queue outbox.

Example 184: *MQ Server with ReplyQueueName Set*

```
<mq:server QueueName="ether" QueueManager="leo"
           ReplyQueueName="outbox" ReplyQueueManager="pager"/>
```

ReplyQueueManager

Overview

`ReplyQueueManager` is mapped to the MQ message descriptor's `ReplyToQMgr` field. It specifies the name of the WebSphere MQ queue manager that controls the reply message queue. When configuring an MQ client port this attribute is required if the clients expect replys to their requests. When configuring an MQ server port you can leave this attribute unset if you are sure that all clients are populating the `ReplyToQMgr` field in the message descriptor of their requests.

Server handling of ReplyQueueManager

When a WebSphere MQ server receives a request, it first looks at the request's message descriptor's `ReplyToQMgr` field. If the request's message descriptor has `ReplyToQ`Mgr set, the server uses the reply queue specified in the message descriptor and ignores the `ReplyQueueManager` setting. If the `ReplyToQMgr` field in the message descriptor is not set, the server will use the `ReplyQueueManager` to determine where to send reply messages.

Example

Example 185 shows a WebSphere MQ client port that is configured to receive replies from the server defined in Example 184 on page 500.

Example 185: *MQ Client with ReplyQueueName Set*

```
<mq:client QueueName="ether" QueueManager="leo"
           ReplyQueueName="outbox" ReplyQueueManager="pager"/>
```

Server_Client

Overview

`Server_Client` specifies which shared libraries to load on systems with a full WebShere MQ installation. Table 39 describes the settings for this attribute for each type of WebShere MQ installation.

Table 39: *Server_Client Settings for the MQ Transport*

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

ModelQueueName

Overview

`ModelQueueName` is only needed if you are using dynamically created queues. It specifies the name of the queue from which the dynamically created queues are created.

AliasQueueName

Overview

When interoperating between WebSphere MQ applications 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 WebSphere MQ client's `AliasQueueName` attribute when you define it in an Artix contract.

Effect of AliasQueueName

When you specify a value for `AliasQueueName` in a WebSphere MQ client port definition, you are altering 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 `ReplyQueueManager` and `ReplyQueueName`. Setting `AliasQueueName` causes Artix to leave `ReplytoQMgr` empty, and to set `ReplyToQ` to the value of `AliasQueueName`. When the `ReplyToQMgr` field of the message descriptor is left empty, the sending queue manager inspects the queue named in the `ReplyToQ` field to determine who its queue manager is and uses that value for `ReplyToQMgr`. The server puts the message on the remote queue that is configured as a proxy for the client's local reply queue.

Example

If you had a system defined similar to that shown in Figure 81, you would need to use the `AliasQueueName` attribute setting when configuring your WebSphere MQ client. In this set up the client is running on a host with a local queue manager `QMgrA`. `QMgrA` has two queues configured. `RqA` is a remote queue that is a proxy for `RqB` and `RplyA` is a local queue. The server is running on a different machine whose local queue manager is `QMgrB`.

QMgrB also has two queues. RqB is a local queue and RplyB is a remote queue that is a proxy for RplyA. The client places its request on RqA and expects replies to arrive on RplyA.

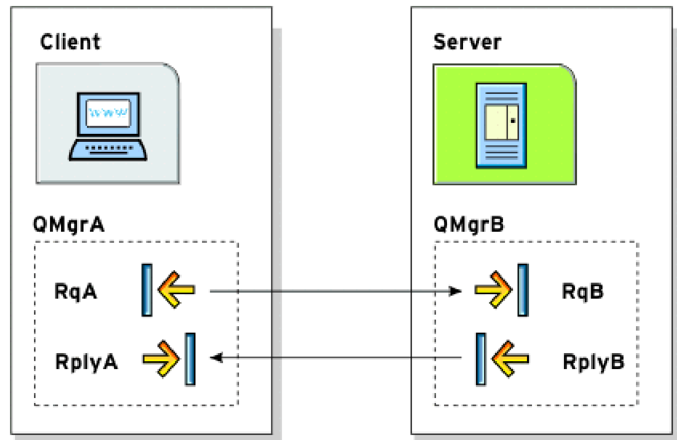


Figure 81: *MQ Remote Queues*

The Artix WebSphere MQ port definitions for the client and server for this deployment are shown in Example 186. AliasQueueName is set to RplyB because that is the remote queue proxying for the reply queue in server's local queue manager. ReplyQueueManager and ReplyQueueName are set to the client's local queue manager so that it knows where to listen for responses. In this example, the server's ReplyQueueManager and ReplyQueueName do not need to be set because you are assured that the client is populating the request's message descriptor with the needed information for the server to determine where replies are sent.

Example 186: *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"/>
```

ConnectionName

Overview

`ConnectionName` specifies the name of the connection Artix uses to connect to its queue.

> **Note:** If you set `CorrelationStyle` to `messageID copy` and specify a value for `ConnectionName` your system will not work as expected.

ConnectionReusable

Overview

`ConnectionReusable` specifies if the connection named in the `ConnectionName` field can be used by more than one application. Valid entries are `true` and `false`. Defaults to `false`.

ConnectionFastPath

Overview

`ConnectionFastPath` specifies if you want to load the request queue manager in process. Valid entries are `true` and `false`. Defaults to `false`.

Example

Example 187 shows a WebSphere MQ client port that loads its request queue manager in process.

Example 187: *WebSphere Client Port using ConnectionFastPath*

```
<mq:client QueueName="gate" QueueManager="dhd"
           ReplyQueueName="inbound" ReplyQueueManager="flipside"
           ConnectionFastPath="true"/>
```

UsageStyle

Overview

`UsageStyle` specifies if a message can be queued without expecting a response. Valid entries are `peer`, `requester`, and `responder`. The default value is `peer`.

Attribute settings

The behavior of each setting is described in Table 40.

Table 40: *UsageStyle Settings*

Attribute Setting	Description
`peer`	Specifies that messages can be queued without expecting any response.
`requester`	Specifies that the message sender expects a response message.
`responder`	Specifies that the response message must contain enough information to facilitate correlation of the response with the original message.

Example

In Example 188, the WebSphere MQ client wants a response from the server and needs to be able to associate the response with the request that generated it. Setting the `UsageStyle` to responder ensures that the server's response will properly populate the response message descriptor's `CorrelID` field according to the defined correlation style. In this case, the correlation style is set to `correlationId`.

Example 188: *MQ Client with UsageStyle Set*

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

CorrelationStyle

Overview

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

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

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

Attribute settings

Table 41 shows the actions of `MQGET` and `MQPUT` when receiving a message using a WSDL specified message ID and a WSDL specified correlation ID.

Table 41: *MQGET and MQPUT Actions*

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

Example

Example 189 shows a WebSphere MQ client application that wants to correlate messages using the `messageID copy` setting.

Example 189: *MQ Client using messageID copy*

```
<mq:client QueueManager="grub" QueueName="gnome"
           ReplyQueueManager="lilo" ReplyQueueName="kde"
           CorrelationStyle="messageId copy"/>
```

AccessMode

Overview

AccessMode controls the action of MQOPEN in the Artix WebSphere MQ transport. Its values can be peek, send, receive, receive exclusive, and receive shared. Each setting mapping corresponds to a WebSphere MQ setting for the MQOPEN. The default is receive.

Attribute settings

Table 42 describes the correlation between the Artix attribute settings and the MQOPEN settings.

Table 42: *Artix WebSphere MQ Access Modes*

Attribute Setting	Description
peek	Equivalent to MQOO_BROWSE. peek opens a queue to browse messages. This setting is not valid for remote queues.
send	Equivalent to MQOO_OUTPUT. send opens a queue to put messages into. The queue is opened for use with subsequent MQPUT calls.
receive (default)	Equivalent to MQOO_INPUT_AS_Q_DEF. receive opens a queue to get messages using a queue-defined default. The default value depends on the DefInputOpenOption queue attribute (MQOO_INPUT_EXCLUSIVE or MQOO_INPUT_SHARED).
receive exclusive	Equivalent to MQOO_INPUT_EXCLUSIVE. receive exclusive opens a queue to get messages with exclusive access. The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open (by this or another application) for input of any type.

Table 42: *Artix WebSphere MQ Access Modes (Continued)*

Attribute Setting	Description
`receive shared`	Equivalent to `MQOO_INPUT_SHARED`. `receive shared` opens queue to get messages with shared access. The queue is opened for use with subsequent `MQGET` calls. The call can succeed if the queue is currently open by this or another application with `MQOO_INPUT_SHARED`.

Example

Example 190 shows the settings for a WebSphere MQ server port that is set-up so that only one application at a time can access the queue.

Example 190: *WebSphere MQ Server setting AccessMode*

```
<mq:server QueueManager="welk" QueueName="anacani"
           ReplyQueueManager="severinsen" ReplyQueueName="johnny"
            AccessMode="recieve exclusive"/>
```

Timeout

Overview

`Timeout` specifies the amount of time, in milliseconds, between a request and the corresponding reply before an error message is generated. If the reply to a particular request has not arrived after the specified period, it is treated as an error.

Example

Example 191 shows the settings for a MQ client port where replies are required in at most 3 minutes.

Example 191:*WebSphere MQ Client Port with a 3 Minute Timeout*

```
<mq:client QueueManager="jpl" QueueName="appollo"
           ReplyQueueManager="jpl" ReplyQueueName="mercury"
           Timeout="180000"/>
```

MessageExpiry

Overview

`MessageExpiry` is mapped to the MQ message descriptor's `Expiry` field. It specifies message lifetime, expressed in tenths of a second. It is set by the Artix endpoint that puts the message onto the queue. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

The value is decremented to reflect the time the message spends on the destination queue, and also on any intermediate transmission queues if the put is to a remote queue. It may also be decremented by message channel agents to reflect transmission times, if these are significant.

`MessageExpiry` can also be set to `INFINITE` which indicates that the messages have unlimited lifetime and will never be eligible for deletion. If `MessageExpiry` is not specified, it defaults to `INFINITE` lifetime.

Example

Example 192 shows the settings for a WebSphere MQ client port where the messages sent from applications using this port have a lifetime of 30 minutes.

Example 192: *Client Port with a 3 Minute Message Lifetime*

```
<mq:client QueueManager="domino" QueueName="dot"
           ReplyQueueManager="domino" ReplyQueueName="cash"
           MessageExpiry="18000"/>
```

MessagePriority

Overview

`MessagePriority` is mapped to the MQ message descriptor's `Priority` fileld. It specifies the message's priority. Its value must be greater than or equal to zero; zero is the lowest priority. If not specified, this field defaults to `priority normal`, which is 5. The special values for `MessagePriority` include `highest` (9), `high` (7), `medium` (5), `low` (3) and `lowest` (0).

Delivery

Overview

`Delivery` can be `persistent` or `not persistent`. `persistent` means that the message survives both system failures and restarts of the queue manager. Internally, this sets the `MQMD`'s `Persistence` field to `MQPER_PERSISTENT` or `MQPER_NOT_PERSISTENT`. The default value is `not persistent`. To support transactional messaging, you must make the messages `persistent`.

Example

Example 193 shows the settings for a WebSphere MQ port that sends persistent oneway messages.

Example 193:*Persistent WebSphere MQ Port*

```
<mq:client QueueManager="mointor" QueueName="msgQ"
           Delivery="persistent"/>
```

Transactional

Overview

`Transactional` controls how messages participate in transactions and what role Webshpere MQ plays in the transactions.

Attribute settings

The values of the `Transactional` attribute are explained in Table 43.

Table 43: *Transactional Attribute Settings*

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

Reliable MQ messages

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

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

Example

Example 194 shows the settings for a WebSphere MQ server port whose requests will be part of transactions managed by WebSphere MQ. Note that the `Delivery` attribute must be set to `persistent` when using transactions.

Example 194: *MQ Client Setup to use Transactions*

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

ReportOption

Overview

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

Attribute settings

The values of this attribute are explained in Table 44.

Table 44: *ReportOption Attribute Settings*

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

Table 44: *ReportOption Attribute Settings (Continued)*

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

Example

Example 195 shows the settings for a WebSphere MQ client that wants to be notified if any of its messages expire before they are delivered.

Example 195: *MQ Client Setup to Receive Expiration Reports*

```
<mq:client QueueManager="herman" QueueName="eddie"
           ReplyQueueManager="gomez" ReplyQueueName="lurch"
           ReportOption="expiration"/>
```

Format

Overview

Format is mapped to the MQ message descriptor's Format field. It specifies an optional format name to indicate to the receiver the nature of the data in the message. The name may contain any character in the queue manager's character set, but it is recommended that the name be restricted to the following:

- Uppercase A through Z
- Numeric digits 0 through 9

Special values

FormatType can take the special values none, string, event, programmable command, and unicode. These settings are described in Table 45.

Table 45: *FormatType Attribute Settings*

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

Table 45: *FormatType Attribute Settings (Continued)*

Attribute Setting	Description
`programmable command`	Corresponds to `MQFMT_PCF`. `programmable command` specifies that the messages are user-defined messages that conform to the structure of a programmable command format (PCF) message. For more information, consult the IBM Programmable Command Formats and Administration Interfaces documentation at http://publibfp.boulder.ibm.com/epubs/html/csqzac03/csqzac030d.htm#Header_12.

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

Example

Example 196 shows a WebSphere MQ client port used for making requests against a server on a mainframe system. Note that the `Convert` attribute is set to `true` signifying that WebSphere will convert the data into the mainframes native data mapping.

Example 196: *WebSphere MQ Client Talking to the Mainframe*

```
<mq:client QueueManager="hunter" QueueName="bigGuy"
           ReplyQueueManager="slate" ReplyQueueName="rusty"
           Format="string" Convert="true"/>
```

MessageId

Overview

`MessageId` is is mapped to the MQ message descriptor's `MsgId` field. It is an alphanumeric string of up to 20 bytes in length. Depending on the setting of `CorrelationStyle`, this string may be used to correlate request and response messages with each other. A value must be specified in this attribute if `CorrelationStyle` is set to `none`.

Example

Example 197 shows the settings for a WebSphere MQ client that wants to use message IDs to correlate response and request messages.

Example 197:*WebSphere MQ Client using MessageID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           CorrelationStyle="messageId" MessageID="foo"/>
```

CorrelationId

Overview

`CorrelationId` is mapped to the MQ message descriptor's `CorrelId` field. It is an alphanumeric string of up to 20 bytes in length. Depending on the setting of `CorrelationStyle`, this string will be used to correlate request and response messages with each other. A value must be specified in this attribute if `CorrelationStyle` is set to `none`.

Example

Example 198 shows the settings for a WebSphere MQ client that wants to use correlation Ids to correlate response and request messages.

Example 198: *WebSphere MQ Client using CorrelationID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
          CorrelationStyle="correlationId" CorrelationID="foo"/>
```

ApplicationData

Overview

`ApplicationData` specifies any application-specific information that needs to be set in the message header.

AccountingToken

Overview

`AccountingToken` is mapped to the MQ message descriptor's `AccountingToken` field. It specifies application-specific information used for accounting purposes.

Example

Example 199 shows the settings for a WebSphere MQ client used for making requests against a server on a mainframe system that keeps tracks of what department is using its resources.

Example 199: *WebSphere MQ Client Sending Accounting Token*

```
<mq:client QueueManager="hunter" QueueName="bigGuy"
           ReplyQueueManager="slate" ReplyQueueName="rusty"
           Format="string" Convert="true"
           AccountingToken="darkHorse"/>
```

Convert

Overview

`Convert` specifies if messages are to be converted to the receiving system's native data format. Valid values are `true` and `false`. Default is `false`.

> **Note:** The WebSphere MQ transport will always attempt to convert string data and always ignore non-string data. This setting is ignored.

Example

Example 200 shows a WebSphere MQ client port used for making requests against a server on a Unix system.

Example 200: *WebSphere MQ Client using Convert*

```
<mq:client QueueManager="atm5" QueueName="ReqQ"
           ReplyQueueManager="hpux1" ReplyQueueName="RepQ"
           Format="string" Convert="true"/>
```

ApplicationIdData

Overview

`ApplicationIdData` is mapped to the MQ message descriptor's `ApplIdentityData` field. It is application-specific string data that can be used to provide additional information about the message or the applciation from which it originated. This attribute is only valid when defining WebSphere MQ clients using an `<mq:client>` element.

ApplicationOriginData

Overview

`ApplicationOriginData` is mapped to the MQ message descriptor's `ApplOriginData` field. It is application-specific string data that can be used to provide additional information about the origin of the message.

Example

Example 201 shows the settings for a WebSphere MQ client that wants to identify itself to the server.

Example 201: *WebSphere MQ Client Sending Origin Data*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           ApplicationOriginData="SSLclient"/>
```

UserIdentification

Overview

`UserIdentification` is mapped to the MQ message descriptor's `UserIdentifier` field. It is a string that represents the User ID of the application from which the message originated. This attribute is only valid when defining Websphere MQ clients using an `<mq:client>` element.

Example

Example 202 shows the settings for a WebSphere MQ client that needs to specify the User that is making the request.

Example 202: *WebSphere MQ Client Sending UserID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           UserIdentification="tux"/>
```

Tibco Port Properties

Artix provides a number of attributes used in defining a TIB/RV service.

Port attributes

Table 46 lists the attributes of the `tibrv:port` element.

Table 46: *TIB/RV Transport Properties*

Attribute	Explanation
`serverSubject`	A required element that specifies the subject to which the server listens. This parameter must be the same between client and server.
`clientSubject`	Specifies the prefix to the subject that the client listens to. The default is to use a uniquely generated name. This parameter only affects clients.
`bindingType`	Specifies the message binding type.

Table 46: *TIB/RV Transport Properties (Continued)*

Attribute	Explanation
callbackLevel	Specifies the server-side callback level when TIB/RV system advisory messages are received.
responseDispatchTimeout	Specifies the client-side response receive dispatch timeout.
transportService	Specifies the UDP service name or port for TibrvNetTransport.
transportNetwork	Specifies the binding network addresses for TibrvNetTransport.
transportDaemon	Specifies the TCP daemon port for the TibrvNetTransport.
transportBatchMode	Specifies if the TIB/RV transport uses batch mode to send messages.
cmSupport	Specifies if Certified Message Delivery support is enabled.
cmTransportServerName	Specifies the server's TibrvCmTransport correspondent name.
cmTransportClientName	Specifies the client TibrvCmTransport correspondent name.
cmTransportRequestOld	Specifies if the endpoint can request old messages on start-up.
cmTransportLedgerName	Specifies the TibrvCmTransport ledger file.

Table 46: *TIB/RV Transport Properties (Continued)*

Attribute	Explanation
cmTransportSyncLedger	Specifies if the endpoint uses a synchronous ledger.
cmTransportRelayAgent	Specifies the endpoint's TibrvCmTransport relay agent.
cmTransportDefaultTimeLimit	Specifies the default time limit for a Certified Message to be delivered.
cmListenerCancelAgreements	Specifies if Certified Message agreements are canceled when the endpoint disconnects.
cmQueueTransportServerName	Specifies the server's TibrvCmQueueTransport correspondent name.
cmQueueTransportWorkerWeight	Specifies the endpoint's TibrvCmQueueTransport `worker weight`.
cmQueueTransportWorkerTasks	Specifies the endpoint's TibrvCmQueueTransport `worker tasks` parameter.
cmQueueTransportSchedulerWeight	Specifies the TibrvCmQueueTransport `scheduler weight` parameter.
cmQueueTransportSchedulerHeartbeat	Specifies the endpoint's TibrvCmQueueTransport `scheduler heartbeat` parameter.
cmQueueTransportSchedulerActivation	Specifies the TibrvCmQueueTransport `scheduler activation` parameter.

Table 46: *TIB/RV Transport Properties (Continued)*

Attribute	Explanation
cmQueueTransportCompleteTime	Specifies the TibrvCmQueueTransport complete time parameter.

bindingType

bindingType specifies the message binding type. TIB/RV Artix ports support three types of payload formats as described in Table 47.

Table 47: *TIB/RV Supported Payload formats*

Setting	Payload Formats	TIB/RV Message Implications
msg	TibrvMsg	The message data is encapsulated in a TibrvMsg described by the binding section of the service's contract.
xml	SOAP, tagged data	The message data is encapsulated in a field of TIBRVMSG_XML with a null name and an ID of 0.
opaque	fixed record length data, variable record length data	The message data is encapsulated in a field of TIBRVMSG_OPAQUE with a null name and an ID of 0.

callbackLevel

callbackLevel specifies the server-side callback level when TIB/RV system advisory messages are received. It has three settings:

- INFO
- WARN
- ERROR (default)

This parameter only affects servers.

responseDispatchTimeout

responseDispatchTimeout specifies the client-side response receive dispatch timeout. The default is TIBRV_WAIT_FOREVER. Note that if only the TibrvNetTransport is used and there is no server return response for a request, then not setting a timeout value causes the client to block forever.

This is because client has no way to know whether any server is processing on the sending subject or not. In this case, we recommend that `responseDispatchTimeout` is set.

transportService

`transportService` specifies the UDP service name or port for TibrvNetTransport. If empty or omitted, the default is `rendezvous`. If no corresponding entry exists in `/etc/services`, `7500` for the `TRDP` daemon, or `7550` for the `PGM` daemon will be used. This parameter must be the same for both client and server.

transportNetwork

`transportNetwork` specifies the binding network addresses for TibrvNetTransport. The default is to use the interface IP address of the host for the `TRDP` daemon, `224.0.1.78` for the `PGM` daemon. This parameter must be interoperable between the client and the server.

transportDaemon

`transportDaemon` specifies the TCP daemon port for TibrvNetTransport. The default is to use `7500` for the `TRDP` daemon, or `7550` for the `PGM` daemon.

transportBatchMode

`transportBatchMode` specifies if the TIB/RV transport uses batch mode to send messages. The default is `false` which specifies that the endpoint will send messages as soon as they are ready. When set to `true`, the endpoint will send its messages in timed batches.

cmSupport

`cmSupport` specifies if Certified Message Delivery support is enabled. The default is `false` which disables CM support. Set this parameter to `true` to enable CM support.

Note: When CM support is disabled all other CM properties are ignored.

cmTransportServerName

`cmTransportServerName` specifies the server's TibrvCmTransport correspondent name. The default is to use a transient correspondent name. This parameter must be the same for both client and server if the client also uses Certified Message Delivery.

cmTransportClientName

`cmTransportClientName` specifes the client's TibrvCMTransport correspondent name. The default is to use a transient correspondent name.

cmTransportRequestOld	`cmTransportRequestOld` specifies if the endpoint can request old messages on start-up. requestOld parameter. The default is `false` which disables the endpoint's ability to request old messages when it starts up. Setting this property to `true` enables the ability to request old messages.
cmTransportLedgerName	`cmTransportLedgerName` specifes the file name of the endpoint's TibrvCMTrasnport ledger. The default is to use an in-process ledger that is stored in memory.
cmTransportSyncLedger	`cmTransportSyncLedger` Specifies if the endpoint uses a synchronous ledger. `true` specifies that the endpoint uses a synchronous ledger. The default is `false`.
cmTransportRelayAgent	`cmTransportRelayAgent` Specifies the endpoint's TibrvCmTransport relay agent. If this property is not set, the endpoint does not use a relay agent.
cmTransportDefaultTimeLimit	`cmTransportDefaultTimeLimit` specifies TibrvCmTransport message default time limit. The default is that no message time limit will be set.
cmListenerCancelAgreements	`cmListenerCancelAgreements` specifies if the TibrvCmListener cancels Certified Message agreements when the endpoint disconnects. parameter. If set to `true`, CM agreements are cancelled when the endpoint disconnects. The default is `false`.
cmQueueTransportServerName	`cmQueueTransportServerName` specifies the server's TibrvCmQueueTransport correspondent name. If this property is set, the server listener joins to the distributed queue of the specified name. This parameter must be the same among the server queue members.
cmQueueTransportWorkerWeight	`cmQueueTransportWorkerWeight` specifies the endpoint's TibrvCmQueueTransport `worker weight`. The default is `TIBRVCM_DEFAULT_WORKER_WEIGHT`.

cmQueueTransportWorkerTasks

cmQueueTransportWorkerTasks specifies the endpoint's TibrvCmQueueTransport worker tasks parameter. The default is TIBRVCM_DEFAULT_WORKER_TASKS.

cmQueueTransportSchedulerWeight

cmQueueTransportSchedulerWeight specifies the TibrvCmQueueTransport scheduler weight parameter. The default is TIBRVCM_DEFAULT_SCHEDULER_WEIGHT.

cmQueueTransportSchedulerHeartbeat

cmQueueTransportSchedulerHeartbeat specifies the TibrvCmQueueTransport scheduler heartbeat parameter. The default is TIBRVCM_DEFAULT_SCHEDULER_HB.

cmQueueTransportSchedulerActivation

cmQueueTransportSchedulerActivation Specifies the TibrvCmQueueTransport scheduler activation parameter. The default is TIBRVCM_DEFAULT_SCHEDULER_ACTIVE.

cmQueueTransportCompleteTime

cmQueueTransportCompleteTime specifies the TibrvCmQueueTransport complete time parameter. The default is 0.

Glossary

B

Binding

A binding associates a specific protocol and data format to operations defined in a `portType`.

Bus

See Service Bus

Bridge

A usage mode in which Artix is used to integrate applications using different payload formats.

C

Collection

A group of related WSDL contracts that can be deployed as one or more physical entities such as Java, C++, or CORBA-based applications. A collection can also be deployed as a switch process.

Connection

An established communication link between any two Artix endpoints.

Contract

An Artix contract is a WSDL file that defines the interface and all connection (binding) information for that interface. In the context of the Artix Designer, this contract is referred to as a *Resource*.

A contract contains two components: logical and physical. The logical contract defines things that are independent of the underlying transport and wire format: `portType`, `operation`, `message`, and `type`.

The physical contract defines the wire format, middleware transport, and service groupings, as well as the mapping between the `portType` and the wire formats, and the buffer layout for fixed formats and extensors. The physical contract defines: `port`, `binding`, and `service`.

CORBA

CORBA (Common Object Request Broker Architecture) defines standards for interoperability and portability among distributed objects, independently of the language in which those objects are written. It is a robust, industry-accepted standard from the OMG (Object Management Group), deployed in thousands of mission critical systems.

CORBA also specifies an extensive set of services for creating and managing distributed objects, accessing them by name, storing them in persistent stores, externalizing their state, and defining ad hoc relationships between them. An ORB is the core element of the wider OMG framework for developing and deploying distributed components.

E

End-point

The runtime deployment of one or more contracts, where one or more transports and its marshalling is defined, and at least one contract results in a generated stub or skeleton (thus an end-point can be compiled into an application).

Extensible Style Sheet Transformation

A set of extensions to the XML style sheet language that describes transformations between XML documents. For more information see the XSLT specification.

H

Host

The network node on which a particular service resides.

M

Marshalling Format

A marshalling format controls the layout of a message to be delivered over a transport. A marshalling format is bound to a transport in the WSDL definition of a port and its binding. A binding can also be specified in a logical contract port type, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

Message

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with defined types. A WSDL message is analogous to a parameter in object-oriented programming.

N

nillable

`nillable` is an attribute of an element that specifies that the element can be left out of an instance of the containing complex type.

O

Operation

A WSDL operation is an abstract definition of the action supported by the service. It is defined in terms of input and output messages. An operation is loosely analogous to a function or method in object-oriented programming, or a message queue or business process.

P

Payload Format

The on-the-wire structure of a message over a given transport. A payload format is associated with a port (transport) in the WSDL file using the binding definition.

Port Type

A WSDL port type is a collection of abstract operations, supported by one or more endpoints. A port type is loosely analogous to a class in object-oriented programming. A port type can be mapped to multiple transports using multiple bindings.

Protocol

A protocol is a transport whose format is defined by an open standard.

R

Resource

A resource can be one of two things:

- A WSDL file that defines the interface of your Artix solution
- A Schema that defines one or more types. This schema can be a stand alone resource or it can define the types within a WSDL contract.

Resources are contained within collections. There can be one or more resources in a collection, and the resources can either be specific to that collection, or shared across several collections (shared resources).

Resources are created either from scratch using the Resource Editor wizards and dialogs to define them, or are based on an existing files. For example, you can use Artix Designer to convert an IDL file into WSDL.

Resource Editor
A GUI tool used for editing Artix resources. It provides several wizards for adding services, transports, and bindings to an Artix resource.

Routing
The redirection of a message from one WSDL binding to another. Routing rules apply to an end-point, and the specification of routing rules is required for a some Artix services. Artix supports topic-, subject- and content-based routing. Topic- and subject-based routing rules can be fully expressed in the WSDL contract. However, content-based routing rules may need to be placed in custom handlers (C plug-ins). Content-based routing handler plug-ins are dynamically loaded.

Router
A usage mode in which Artix redirects messages based on rules defined in an Artix contract.

S

Service
An Artix service is instance of an Artix runtime deployed with one or more contracts, but no generated language bindings (contrast this with end-point). The service acts as a daemon that has no compile-time dependencies. A service is dynamically configured by deploying one or more contracts on it.

Service Bus
The infrastructure that allows service providers and service consumers to interact in a distributed environment. Handles the delivery of messages between different middleware systems. Also known as an Enterprise Service Bus.

SOAP
SOAP is an XML-based messaging framework specifically designed for exchanging formatted data across the Internet. It can be used for sending request and reply messages or for sending entire XML documents. As a protocol, SOAP is simple, easy to use, and completely neutral with respect to operating system, programming language, or distributed computing platform.

Standalone Mode

An Artix instance running independently of either of the applications it is integrating. This provides a minimally invasive integration solution and is fully described by an Artix contract.

Switch

The implementation of an Artix WSDL service contract.

System

A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-In

A plug-in module that provides wire-level interoperation with a specific type of middleware. When configured with a given transport plug-in, Artix will interoperate with the specified middleware at a remote location or in another process. The transport is specified in the 'Port' property in of an Artix contract.

Type

A WSDL data type is a container for data type definitions that is used to describe messages (for example an XML schema).

W

Web Services Description Language

An XML based specification for defining Web services. For more information see the WSDL specification.

WSDL

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data binding formats. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract

collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- **Types**—a container for data type definitions using some type system.
- **Message**—an abstract, typed definition of the data being communicated.
- **Operation**—an abstract definition of an action supported by the service.
- **Port Type**—an abstract set of operations supported by one or more endpoints.
- **Binding**—a concrete protocol and data format specification for a particular port type.
- **Port**—a single endpoint defined as a combination of a binding and a network address.
- **Service**—a collection of related endpoints.

Source: Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001. (http://www.w3.org/TR/wsdl)

X

XML

XML is a simpler but restricted form of SGML (Standard General Markup Language). The markup describes the meaning of the text. XML enables the separation of content from data. XML was created so that richly structured documents could be used over the web.

XSD

XML Schema Definition (XSD) is the language used to define an XML Schema. The XML Schema defines the structure of an XML document.

In Artix, a schema can be a standalone resource within a collection, or it can be used as an import do define the types within a WSDL contract.

Index

A
adding
  IDL 41
Address specification
  CORBA 265
  HTTP 255
  IIOP 275
  SOAP 255
arrays
  CORBA 418

B
bindings
  adding
    CORBA 153
    fixed 166
    FML 164
    SOAP 138
    SOAP with Attachments 149
    tagged 186
    XML 237
  mapping
    CORBA 155
    fixed 170
    FML field tables 160
    G2++ 243
    tagged 191, 204
    XML 239
bus-security 328

C
C++ code generation preferences 15
coboltowsdl 67
complexContent 433
complexType 425
configuring IIOP 275
Connecting to remote queues 504
corba:address 265
corba:alias 417
corba:anonsequence 426
corba:array 418
corba:binding 155
corba:case 416
corba:corba_input_attributes 328
corba:enum 411
corba:enumerator 411
corba:excepetion 423
corba:fixed 412
corba:member 409, 423
corba:object 441
corba:operation 155
corba:param 155
corba:policy 266
corba:raises 156
corba:return 156
corba:struct 409
corba:union 415
corba:unionbranch 415

E
enumerations
  CORBA 411
exceptions
  CORBA 423
extension 433

F
fixed:binding 171
fixed:body 171
fixed:enumeration 175
  fixedValue 175
  value 175
fixed:field 173
  bindingOnly 172
  fixedValue 175
  format 173
  size 173
fixed:operation 171
fixed:sequence 179
fixed data types
  CORBA 412

G
generating contracts

from COBOL 62
from IDL 40
from Java 50

H

HTTP
Accept 467
AcceptEncoding 469
AcceptLanguage 468
Authorization 466
AuthorizationType 465
AutoRedirect 462
BrowserType 479
CacheControl 474
ClientCertificate 455
ClientCertificateChain 455
ClientPrivateKey 485
ClientPrivateKeyPassword 455
Connection 473
ConnectionAttempts 454
ContentEncoding 471
ContentLocation 457
ContentType 470
Cookie 478
HonorKeepAlive 488
Host 472
Password 464
ProxyAuthorization 483
ProxyAuthorizationType 482
ProxyPassword 455
ProxyServer 481
ProxyUserName 454
ReceiveTimeout 461
RedirectURL 489
Referer 480
SendTimeout 460
ServerCertificate 457
ServerCertificateChain 491
ServerPrivateKey 457
ServerPrivateKeyPassword 458
ServerType 490
SuppressClientReceiveErrors 487
SuppressClientSendErrors 486
TrustedRootCertificate 455, 458
UserName 463
UseSecureSockets 484
http:address 255
http-conf:HTTPServerIncomingContexts 328

I

i18n-context:client 310
InboundCodeSet 310
LocalCodeSet 310
OutboundCodeSet 310
i18n-context:server 310
InboundCodeSet 310
LocalCodeSet 310
OutboundCodeSet 310
idltowsdl 43
ignorecase 328
iiop:address 275
iiop:payload 275
iiop:policy 275
IOR specification 265, 275

J

Java code generation preferences 14
javatowsdl 50
jms:address
durableSubscriberName 290
messageSelector 290
transactional 290
useMessageIDAsCorrelationID 290

M

mime:content 149
mime:multipartRelated 148
mime:part 148
mq:client 279
mq:server 279
MQ remote queues 504

N

nillable 436

P

portType 128

R

route
creating 330
routing
broadcast 325
failover 326
fanout 325
load balancing 338

routing:contains 329
routing:destination 320
port 320
service 320
routing:empty 329
routing:endswith 329
routing:equals 328
routing:equals:contextAttributeName 328
routing:equals:contextName 328
routing:equals:value 328
routing:greater 328
routing:less 328
routing:nonempty 329
routing:operation 322
name 322
target 322
routing:route 319
multiRoute 325, 326, 338
failover 326
fanout 325
loadBalance 338
name 319
routing:source 319
port 319
service 319
routing:startswith 328
routing:transportAttribute 327

S

soap:address 255
soap:body
parts 143
soap:header 142
encodingStyle 142
message 142
namespace 142
part 142
use 142
soapenc:base64 428
Specifying POA policies 266, 275
structures
CORBA 409

T

tagged:binding 192
tagged:body 194
tagged:case 197
tagged:choice 197
tagged:enumeration 194
tagged:field 194
tagged:operation 193
tagged:sequence 195
tibrv:array 211
tibrv:binding 205
tibrv:context 234
tibrv:field 232
tibrv:input 206
tibrv:msg 231
tibrv:operation 206
tibrv:output 207
tibrv:port 298
tibrv:port@bindingType 536
tibrv:port@callbackLevel 536
tibrv:port@clientSubject 533
tibrv:port@cmListenerCancelAgreements 538
tibrv:port@cmQueueTransportCompleteTime 539
tibrv:port@cmQueueTransportSchedulerActivation 539
tibrv:port@cmQueueTransportSchedulerHeartbeat 539
tibrv:port@cmQueueTransportSchedulerWeight 539
tibrv:port@cmQueueTransportServerName 538
tibrv:port@cmQueueTransportWorkerTasks 539
tibrv:port@cmQueueTransportWorkerWeight 538
tibrv:port@cmSupport 537
tibrv:port@cmTransportClientName 537
tibrv:port@cmTransportDefaultTimeLimit 538
tibrv:port@cmTransportLedgerName 538
tibrv:port@cmTransportRelayAgent 538
tibrv:port@cmTransportRequestOld 538
tibrv:port@cmTransportServerName 537
tibrv:port@cmTransportSyncLedger 538
tibrv:port@serverSubject 533
tibrv:port@transportBatchMode 537
tibrv:port@transportDaemon 537
tibrv:port@transportNetwork 537
tibrv:port@transportService 537
tuxedo:binding 164
tuxedo:field 164
tuxedo:fieldTable 164
tuxedo:input 305
tuxedo:operation 164
tuxedo:server 305
tuxedo:service 305
typedefs
CORBA 417

U

unions
  Artix mapping 414
  CORBA 414, 415
  logical description 414

W

WebSphere MQ
  AccessMode 512
  AccountingToken 527
  AliasQueueName 504
  ApplicationData 526
  ApplicationIdData 529
  ApplicationOriginData 530
  ConnecitonName 506
  ConnectionFastPath 508
  ConnectionReusable 507
  Convert 528
  CorrelationId 525
  CorrelationStyle 510
  Delivery 517
  Format 522
    working with mainframes 523
  MessageExpiry 515
  MessageId 524
  MessagePriority 516
  ModelQueueName 503
  QueueManager 498
  QueueName 499
  ReplyQueueManager 501
  ReplyQueueName 500
  ReportOption 520
  Server_Client 502
  Timeout 514
  Transactional 518
  UsageStyle 509
  UserIdentification 531
wsdltocorba 154, 271
wsdltoservice
  adding a CORBA service 269
  adding a JMS service 294
  adding an HTTP service 258
  adding an IIOP service 276
  adding a TIBCO service 301
  adding a Tuxedo service 307
  adding a WebSphere MQ service 282
wsdltosoap 138, 139

X

xformat:binding 239
  rootNode 239
xformat:body 239
  rootNode 239
XML Schema 95
XML Stylesheet Language Transformations 380
XPath 383
XSD 95
xsd:annotation 440
xsd:base64Binary 428
xsd:hexBinary 428
xsdtowsdl 78
xsl:apply-templates 385
  select 386
xsl:copy-of 389
  select 389
xsl:element 389
  name 389
  namespace 389
xsl:stylesheet 381
xsl:template 383
  match 383
xsl:transform 381
xsl:value-of 389
  select 389
XSLT 380