Artix™

Designing Artix Solutions from the Command Line

Version 2.0, March 2004

Making Software Work Together™

Updated: 30-Apr-2004

M 3 1 8 9

Contents

List of Figures

List of Tables

Preface

What is Covered in this Book

Designing Artix Solutions from the Command Line provides the reader with detailed information about how to design Artix solutions and describe those solutions in Artix contracts. It begins with an overview of the conceptes needed by a user of Artix and a description of WSDL. It hen moves into detialed descriptions of the Artix WSDL extentions used to describe each of the transports and payload formats supported by Artix. These detailed descriptions cover how complex data types are mapped to into a paylaod format and how to provide specific configuration information for particular transports.

Note: This book does not provide descriptions or information about the supported transports. For information on how to set-up and use them see the documentation provided by the vendors.

In addition, this book covers all of the command line tools provided with Artix to assist you in building your Artix contracts. These include tools to convert IDL to WSDL, tools to add CORBA bindings to existing Artix contracts, and others.

Who Should Read this Book

The target audience for *Designing Artix Solutions from the Command Line* is the designer of Artix solutions who wants an understanding of the internals of Artix contracts. The reader should have a working knowledge of the middleware transports that are being used to implement the Artix soultion.

How to Use this Book

If you are new to Artix and WSDL, the first three chapters of this book provide overviews of Artix and WSDL. "Introduction to Using Artix" provides an overview of the concepts behind using Artix to solve integration projects. "Understanding WSDL" describes the basics of Web Services Description Language and how to map services. "Understanding Artix Contracts" describes how Artix extends WSDL to describe transport independent services and integration. A working knowledge of this information is helpfull in understanding the content of the following chapters which deal with specific middleware products, transports, and payload formats.

If you are interested in adding routing information to you Artix solution, Chapter 4 describes how to create message routes in an Artix contract.

To learn about how Artix interacts with the major middleware products it can integrate, you will want to read one or more of the following chapters:

- Chapter 6 describes how to integrate CORBA systems into an Artix solution.
- Chapter 7 describes how to integrate BEA Tuxedo in an Artix solution.

  Note: BEA Tuxedo integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports BEA Tuxedo integration.

- Chapter 8 describes how to integrate TIBCO Rendezvous into an Artix solution.

  Note: TIBCO Rendezvous integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports TIBCO Rendezvous integration.

- Chapter 9 describes how to integrate IBM WebSphere MQ systems into an Artix solution.

  Note: IBM WebSphere MQ integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports IBM WebSphere MQ integration.

- Chapter 10 describes how to use Artix with the Java Messaging System.

  Note: Java Messaging System integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports Java Messaging System integration.

These chapters are focused on describing data and port configurations using Artix. They do not provide details about the middleware products beyond what is related to making Artix solutions interact with them.

If you are using artix with transports and paylaod formats that have open standards, you will want to read one or more of the following:

- Chapter 11 describes how to use HTTP with Artix.
- Chapter 12 describes how to use the IIOP tunnel transport.

  Note: The IIOP tunnel transport is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports the IIOP tunnel transport.

- Chapter 13 describes how to use SOAP messages in Artix.
- Chapter 14 describes how to use fixed record length data in Artix.
- Chapter 15 describes how to use self-describing messages in Artix.
- Chapter 16 describes how to use the G2++ and XML payload formats supported by Artix.

Online Help

While using the Artix Designer you can access contextual online help, providing:

- A description of your current Artix Designer screen
- Detailed step-by-step instructions on how to perform tasks from this screen
- A comprehensive index and glossary
- A full search feature

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

Finding Your Way Around the Artix Librabry

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

If you are new to Artix

You may be interested in reading:

- *Getting Started with Artix* - the Getting Started books (Encompass, Relay, and Java) describe basic Artix concepts. These books also provide a walk through Artix to solve a real world problem using code provided in the product kit.
- *Artix Tutorial* - this book guides you through programming Artix applications against all of the supported transports.

To design Artix solutions

You should read one or more of the following:

- *Designing Artix Solutions* - this book provides detailed information about using the Artix Designer to create WSDL based Artix contracts, Artix stub and skeleton code, and Artix deployment descriptors.
- *Designing Artix Solutions from the Command Line* - this book provides detailed information about the WSDL extensions used in Artix contracts, and explains the mappings between data types and Artix bindings.

To develop applications using Artix stub and skeleton code

Depending on your development environment you should read one or more of the following:

- *Developing Artix Applications in C++* - this book discusses the technical aspects of programming applications using the Artix C++ API
- *Developing Artix Applications in Java* - this book discusses the technical aspects of programming applications using the Artix Java API

To manage and configure your Artix solution

You should read *Deploying and Managing Artix Solutions.* It describes how to configure and deploy Artix-enabled systems. It also discusses how to manage them once they are deployed.

If you want to know more about Artix security

You should read the *Artix Security Guide*. It outlines how to enable and configure Artix's security features. It also discusses how to integrate Artix solutions into a secure environment.

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 details provided there with the last updated date printed on the inside cover of the book you are using (at the bottom of the copyright notice).

Additional Resources for Help

The IONA knowledge base contains helpful articles, written by IONA experts, about Artix and other products. You can access the knowledge base at the following location:

The IONA update center contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical Conventions

This book uses the following typographical conventions:

`Constant width`

Constant 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 `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Italic	Italic words in normal text represent *emphasis* and *new terms*. 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/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	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 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.
|	A vertical bar separates items in a list of choices enclosed in {} (braces) in format and syntax descriptions.

CHAPTER 1

Introduction to Using Artix

Artix allows you to design and deploy integration solutions that are middleware-neutral.

In this chapter

This chapter discusses the following topics:

The Artix Bus

Overview

The Artix bus provides a middleware connectivity solution that minimizes invasiveness and lets an organization avoid being locked into any one middleware transport. For example, the Artix bus can be used to connect a BEA Tuxedo™-based server to a CORBA client. The Artix bus transparently handles the message mapping and transformation between them. The Tuxedo server is unaware that its client is using CORBA. In fact, with the bus handling the communication, the client could be changed to an IBM WebSphere MQ™ client without modifying the server.

Bus message transporting

The Artix bus shields applications from the details of the transports used by applications on the other end of the bus, by providing on-the-wire message transformation and mapping. Unlike the approach taken by Enterprise Application Integration (EAI) products, the Artix bus does not use an intermediate canonical format; it transforms the messages once. Figure 1 shows a high level view of how a message passes through the bus.

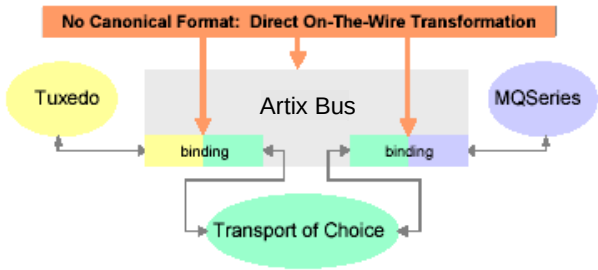

Figure 1: *Artix Message Transporting*

The approach taken by the Artix bus provides a high level of throughput by avoiding the overhead of making two transformations for each message. The approach does, however, limit the flexibility of message mapping. The Artix bus can only map messages across varying transports; it cannot modify the content or structure of the message.

Supported message transports

The Artix bus supports the following message transports:

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

Supported payload formats

The Artix bus can automatically transform between the following payload formats:

- G2++
- FML – Tuxedo format
- CORBA (GIOP) – CORBA format
- FRL – fixed record length
- VRL – variable record length
- SOAP
- TibrvMsg - TIBCO Rendezvous format

Bus contracts

An Artix bus contract defines the interaction of a Service Access Point (SAP) or endpoint with an Artix bus. Contracts are written using a superset of the standard Web Service Definition Language (WSDL). Following the procedure described by W3C, IONA has extended WSDL to support the bus' advanced functionality, and use of transports and formats other than HTTP and SOAP.

A bus contract consists of two parts:

Logical

The logical portion of the contract defines the namespaces, messages, and operations that the SAP exposes. This part of the contract is independent of the underlying transports and wire formats. It fully specifies the data structures and possible operation/interaction with the interface. It is made up of the WSDL tags `<message>`, `<operation>`, and `<portType>`.

Physical

The physical portion of the contract defines the transports, wire formats, and routing information used to deliver messages to and from SAPs, over the bus. This portion of the contract also defines which messages use each

of the defined transports and bindings. The physical portion of the contract is made up of the standard WSDL tags `<binding>`, `<port>`, and `<operation>`. It is also the portion of the contract that may contain IONA WSDL extensions.

Deployment models

Applications that use the Artix bus can be deployed in one of two ways:

Embedded mode is the most invasive use of the Artix bus and provides the highest performance. In embedded mode, an application is modified to invoke Artix functions directly and locally, as opposed to invoking a standalone Artix service. This approach is the most invasive to the application, but also provides the highest performance. Embedded mode requires linking the application with Artix-generated stubs and skeletons to connect client and server (respectively) to the Bus.

Standalone mode runs as a separate process invoked as a service. In standalone mode, the Artix bus provides a zero-touch integration solution on the application side. When designing a system, you simply generate and deploy the Artix contracts that specify each endpoint of the bus. Because a standalone switch is not linked directly with the applications that use it (as in embedded mode), a contract for standalone mode deployment must specify routing information. This is the least efficient of the two modes.

Advanced Features

The Artix bus also supports the following advanced functionality:

- Message routing based on the operation or the port, including routing based on characteristics of the port.
- Transaction support over Tuxedo and WebSphere MQ.
- SSL and TLS support.
- Security support for Tuxedo and WebSphere MQ.
- Container based deployment with IONA's Application Server Platform 6.0 and Tuxedo 7.1 or higher.

The Artix Design Process

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 Web services or as a service for any number of applications using other middleware transports. In addition, Artix provides a flexible programming model that allows you to create new applications that can communicate using any of protocols that Artix supports.

Despite the flexibility and power of Artix, designing solutions using Artix is a straightforward process which requires a minimum of coding. The Artix Designer provides a full suite of wizards to guide you through the modeling of your systems, the generation of Artix components, and the deployment of your system. Artix also ships with a number of command line tools that can be used to generate Artix components.

Regardless of the complexity of your Artix project or the tools you chose to develop your Artix project, there are four basic steps in developing a solution using Artix:

1. Create an Artix contract to model your existing services.
2. Modify your Artix contract to describe how you intend to integrate or expose your systems.
3. Generate the Artix components.
4. Develop any application level code needed to complete the solution.

Creating an Artix contract

The first step in solving a problem using Artix is to create a contract which models the services you want to integrate. This involves creating logical descriptions of the data and the operations you want the services to share, and mapping them to the physical payload formats and transports the services use to expose themselves to the network. Artix uses the industry standard Web Services Description Language (WSDL) to model services.

For more information on Artix contracts and modeling services in WSDL, read “Understanding WSDL” on page 7.

Describe the integration of the services

After describing how your services are currently deployed, you must decide how you want them to be integrated. If your services share a common interface, you may simply need to add routing rules to your contract. Artix provides a rich set of routing capabilities to map operations and interfaces to one another. For a detailed discussion of routing, see Chapter 4 on page 27.

If you are exposing an existing service using a new transport or payload format, you need to add the mapping of the service's data and operations to the new payload format and transport.

Generate Artix components

If you are using Artix in standalone mode, you will need to generate a configuration scope for your Artix switch and save the Artix contract defining the interaction of your services.

If you are using Artix in embedded mode, you will also need to generate the Artix stubs and skeletons that will form the backbone of your Artix application code.

For a detailed discussion of Artix configuration, see the *Artix Administration Guide*. For a detailed description of generating Artix stubs and skeletons, see the *Artix C++ Programmer's Guide*.

Develop application code

Unless your services share identical interfaces, you will need to develop some application code. Artix can only map between services that share a common interface. Typically, you can make the required changes to only one side of the services you are integrating and you can write the application code using a familiar programming paradigm. For example, if you are a CORBA developer integrating a CORBA system with a Tuxedo application, Artix will generate the IDL representing the interface used in the service integration. You can then implement the interface using CORBA.

If you are developing new applications using Artix, you will have to write the application logic from scratch using the stubs and skeletons generated by Artix. For a detailed discussion of developing applications using Artix, see the *Artix C++ Programmer's Guide*.

CHAPTER 2

Understanding WSDL

Artix contracts are WSDL documents that describe logical services and the data they use.

In this chapter

This chapter discusses the following topics:

Web Services Description Language Basics

Overview

Web Services Description Language (WSDL) is an XML document format used to describe services offered over the Web. WSDL is standardized by the World Wide Web Consortium (W3C) and is currently at revision 1.1. You can find the standard on the W3C website, www.w3.org.

Web service endpoints and Artix service access points

WSDL documents describe a service as a collection of *endpoints*. Each endpoint is defined by binding an abstract operation description to a concrete data format and specifying a network protocol and address for the resulting binding.

Artix *service access points* extend the concept of endpoint to include services that are available over any computer network, not just the web. A service access point can be bound to payload formats other than SOAP and can use transports other than HTTP.

Abstract operations

The abstract definition of operations and messages is separated from the concrete data formatting definitions and network protocol details. As a result, the abstract definitions can be reused and recombined to define several endpoints. For example, a service can expose identical operations with slightly different concrete data formats and two different network addresses. Or, one WSDL document could be used to define several services that use the same abstract messages.

Port types

A *portType* is a collection of abstract operations that define the actions provided by an endpoint. When a port type is mapped to a concrete data format, the result is a concrete representation of the abstract definition, in the form of an endpoint or service access point.

Concrete details

The mapping of a particular port type to a concrete data format results in a reusable *binding*. A *port* is defined by associating a network address with a reusable binding, and a collection of ports define a *service*.

Because WSDL was intended to describe services offered over the Web, the concrete message format is typically SOAP and the network protocol is typically HTTP. However, WSDL documents can use any concrete message format and network protocol. In fact, Artix contracts bind operations to several data formats and describe the details for a number of network protocols.

Namespaces and imported descriptions

WSDL supports the use of XML namespaces defined in the `<definition>` element as a way of specifying predefined extensions and type systems in a WSDL document. WSDL also supports importing WSDL documents and fragments for building modular WSDL collections.

Elements of a WSDL document

A WSDL document is made up of the following elements:

- `<types>` – the definition of complex data types based on in-line type descriptions and/or external definitions such as those in an XML Schema (XSD).
- `<message>` – the abstract definition of the data being communicated.
- `<operation>`– the abstract description of an action.
- `<portType>` – the set of operations representing an absract endpoint.
- `<binding>` – the concrete data format specification for a port type.
- `<port>` – the endpoint defined by a binding and a physical address.
- `<service>` – a set of ports.

Example

Example 1 shows a simple WSDL document. It defines a SOAP over HTTP service access point that returns the date.

Example 1: *Simple WSDL*

```
<?xml version="1.0"?>
<definitions name="DateService"
   targetNamespace="urn:dateservice"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="urn:dateservice"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://iona.com/dates/schemas">
```

Example 1: *Simple WSDL*

```
  <types>
    <schema targetNamespace="http://iona.com/dates/schemas"
   xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="dateType">
         <complexType>>
          <all>
            <element name="day" type="xsd:int"/>
            <element name="month" type="xsd:int"/>
            <element name="year" type="xsd:int" />
           </all>
         </complexType>
      <element>
    </schema>
  </types>
  <message name="DateResponse">
    <part name="date" element="xsd1:dateType" />
  </message>
  <portType name="DatePortType">
    <operation name="sendDate">
      <output message="tns:DateResponse" name="sendDate" />
    </operation>
  </portType>
  <binding name="DatePortBinding" type="tns:DatePortType">
    <soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="sendDate">
      <soap:operation soapAction="" style="rpc" />
      <output name="sendDate">
        <soap:body
   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   namespace="urn:dateservice" use="encoded" />
      </output>
    </operation>
  </binding>
  <service name="DateService">
    <port binding="tns:DatePortBinding" name="DatePort">
      <soap:address location="http://www.iona.com/DatePort/" />
    </port>
  </service>
</definitions>
```

Abstract Data Type Definitions

Overview

Applications typically use datatypes that are more complex than the primitive types, like `int`, defined by most programming languages. WSDL documents represent these complex datatypes using a combination of schema types defined in referenced external XML schema documents and complex types described in `<types>` elements.

Complex type definitions

Complex data types are described in a `<types>` element. The W3C specification states the XSD is the preferred canonical type system for a WSDL document. Therefore, XSD is treated as the intrinsic type system. Because these data types are abstract descriptions of the data passed over the wire and not concrete descriptions, there are a few guidelines on using XSD schemas to represent them:

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

WSDL does allow for the specification and use of alternative type systems within a document.

Example

The structure, `personalInfo`, defined in Example 2, contains a `string`, an `int`, and an `enum`. The `string` and the `int` both have equivalent XSD types and do not require special type mapping. The enumerated type `hairColorType`, however, does need to be described in XSD.

Example 2: *personalInfo*

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

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

Example 3 shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in Example 2. `hairColorType` is described using a named `simpleType` because it does not have any child elements. `personalInfo` is defined as an `element` so that it can be used in messages later in the contract.

Example 3: *XSD type definition for personalInfo*

```
<types>
  <xsd:schema targetNamespace="http:\\iona.com\personal\schema"
   xmlns:xsd1="http:\\iona.com\personal\schema"
   xmlns="http://www.w3.org/2000/10/XMLSchema">
    <simpleType name="hairColorType">
      <restriction base="xsd:string">
        <enumeration value="red" />
        <enumeration value="brunette" />
        <enumeration value="blonde" />
      </ restriction>
    </ simpleType>
    <element name="personalInfo">
      <complexType>
        <element name="name" type="xsd:string" />
        <element name="age" type="xsd:int" />
        <element name="hairColor" type="xsd1:hairColorType" />
      </ complexType>
    </ element>
  </ schema>
</ types>
```

Another way to map personalInfo is to describe `hairColorType` in-line as shown in Example 4. WIth this mapping, however, you cannot reuse the description of `hairColorType`.

Example 4: *Alternate XSD mapping for personalInfo*

```
<types>
  <xsd:schema targetNamespace="http:\\iona.com\personal\schema"
   xmlns:xsd1="http:\\iona.com\personal\schema"
   xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="personalInfo">
      <complexType>
        <element name="name" type="xsd:string" />
        <element name="age" type="xsd:int" />
```

Example 4: *Alternate XSD mapping for personalInfo*

```
        <element name="hairColor">
          <simpleType>
            <restriction base="xsd:string">
              <enumeration value="red" />
              <enumeration value="brunette" />
              <enumeration value="blonde" />
            </ restriction>
          </ simpleType>
        </ element>
      </ complexType>
    </ element>
  </ schema>
</ types>
```

Abstract Message Definitions

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 as defined in Example 2 on page 11 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 5.

Example 5: *personalInfo lookup method*

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

This method signature could be mapped to the WSDL fragment shown in Example 6.

Example 6: *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, requests, or output messages, responses.

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 1

Table 1: *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 7.

Example 7: *Reused part*

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

Example 7: *Reused part*

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

Abstract Interface Definitions

Overview

WSDL <portType> elements define, in an abstract way, the operations offered by a service. The operations defined in a port type list the input, output, and any fault messages used by the service to complete the transaction the operation describes.

Port types

A portType can be thought of as an interface description and in many Web service implementations there is a direct mapping between port types and implementation objects. Port types are 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.

Port types are described using the <portType> element in a WSDL document. Each port type 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 within a port type must have a unique name, specified using the name attribute. The name attribute is required to define an operation.

Elements of an operation

Each operation is made up of a set of elements. The elements represent the messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in Table 2.

Table 2: *Operation Message Elements*

Element	Description
<input>	Specifies a message that is received from another endpoint. This element can occur at most once for each operation.

Table 2: *Operation Message Elements*

Element	Description
`<output>`	Specifies a message that is sent to another endpoint. This element can occur at most once for each operation.
`<fault>`	Specifies a message used to communicate an error condition between the endpoints. This element is not required and can occur an unlimited number of times.

An operation is required to have at least one `input` or `output` element. The elements are defined by two attributes listed in Table 3.

Table 3: *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 port type 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 the binding section.

Example

For example, in implementing a server that stored personal information in the structure defined in Example 2 on page 11, you might use an interface similar to the one shown in Example 8.

Example 8: *personalInfo lookup interface*

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

This interface could be mapped to the port type in Example 9.

Example 9: *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>
```

Mapping to the Concrete Details

Overview

The abstract definitions in a WSDL document are intended to be used in defining the interaction of real applications that have specific network addresses, use specific network protocols, and expect data in a particular format. To fully define these real applications, the abstract definitions need to be mapped to concrete representations of the data passed between the applications and the details of the network protocols need to be added.

This is done by the WSDL bindings and ports. WSDL binding and port syntax is not tightly specified by W3C. While there is a specification defining the mechanism for defining the syntaxes, the syntaxes for bindings other than SOAP and network transports other than HTTP are not bound to a W3C specification.

Bindings

To define an endpoint that corresponds to a running service, port types are mapped to bindings which describe how the abstract messages defined for the port type map to the data format used on the wire. The 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.

Services

The final piece of information needed to describe how to connect a remote service is the network information needed to locate it. This information is 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 many 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.

CHAPTER 3

Understanding Artix Contracts

Artix contracts are WSDL documents that have IONA-specific WSDL extensions, and which define Artix applications.

In this chapter

This chapter discusses the following topics:

Artix Contract Overview

Overview

Artix contracts are WSDL documents that describe Artix service access points and their integration. Each mapping of a port type to a binding and port defines an Artix service access point. An Artix contract also describes the routing between service access points.

An Artix contract has two sections as shown in Figure 2:

Logical describes the abstract operations, messages, and data types used by a service access point.

Physical describes the concrete message formats and transports used by a service access point. The routing information defining how messages are mapped between different service access points is also specified here.

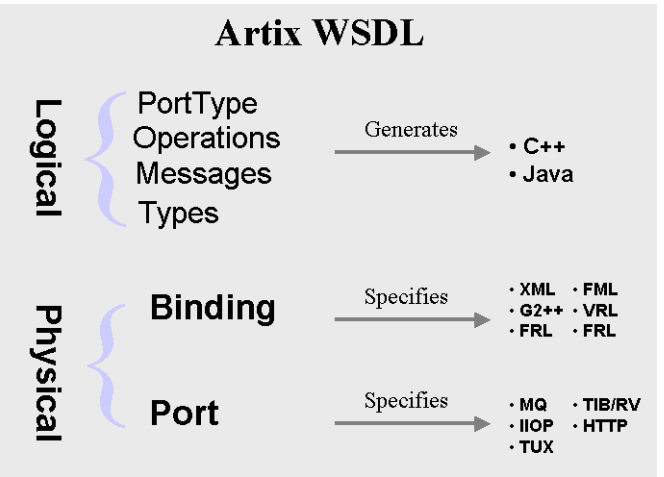

Figure 2: *An Artix Contract*

The Logical Section

Overview

The logical section of an Artix contract defines the abstract operations that the service access points offer. The logical view includes the <types>, <message>, and <portType> tags in a WSDL document. This portion of the contract also specifies the namespaces used in defining the contract.

Namespaces

Artix contracts use several IONA-specific namespaces to define the Artix extensions for mapping to different data formats and network transports. These namespaces include:

Table 4: *Artix Namespaces*

Namespace	Description
http://schemas.iona.com/transports/http	Specifies the WSDL extensions for HTTP
http://schemas.iona.com/transports/http/configuration	Specifies additional extensions to configure the HTTP transport.
http://schemas.iona.com/bindings/corba	Specifies the WSDL extensions used to map data to CORBA. This namespace also specifies the transport specific configuration setting for a CORBA port.
http://schemas.iona.com/bindings/corba/typemap	Specifies the type mapping information used to fully describe complex CORBA types defined in IDL.
http://schemas.iona.com/bindings/fixed	Specifies the WSDL extentions used to describe fixed data bindings.
http?schemas.iona.com/bindings/tagged	Specifies the WSDL extentions used to describe tagged data bindings.
http://schemas.iona.com/routing	Specifies the WSDL extensions to define routing between Artix SAPs.
http://schemas.iona.com/transports/jms	Specifies the WSDL extentions used to describe a JMS port.

Table 4: *Artix Namespaces*

Namespace	Description
`http://schemas.iona.com/transports/mq`	Specifies the WSDL extensions to configure the WebSphere MQ transport.

Port types and code generation

The Artix code generation tools, including the IDL generator, are driven by the port types defined in an Artix contract. For each port type defined in a contract, the code generators create an object named for the port type it represents. For example, the port type defined in Example 9 on page 19 results in an object similar to the one shown in Example 10.

Example 10: *personalInfo Object*

```
class personalInfoLookup
{
  personalInfoLookup();
  ~personalInfoLookup();

  void lookup(int empID, personalLookupResponse &return);
}
```

For more information on Artix code generation, see *Developing Artix Applications in C++* and *Developing Artix Applications in Java*.

The Physical Section

Overview

The physical section of an Artix contract defines the actual bindings and transports used by the service access points. It includes the information specified in the <binding> and <service> tags of a WSDL document. It also includes the routing rules defining how the messages are routed between the endpoints defined in the contract.

Bindings

WSDL is intended to describe service offered over the Web and therefore most bindings are specified using SOAP as the message format. WSDL can bind data to other message formats however.

Artix provides bindings for several message formats including CORBA and FML. For specific information on using these bindings see the appropriate chapter in this guide.

Network protocols

WSDL documents typically use HTTP as the network protocol. However, WSDL is not limited to representing connections over HTTP. Artix provides port descriptions for several network protocols including IIOP and WebSphere MQ. For more information on using these network protocols in Artix see the appropriate chapter in this guide.

CORBA type map

When using the CORBA additional data is required to fully map the logical types to concrete CORBA data types. This is done using a CORBA type map extension to standard WSDL. For a detailed description of how Artix maps logical types to CORBA types read "CORBA Type Mapping" on page 58.

Routing

To fully describe the integration of service access points across an enterprise, Artix contracts include routing rules for directing data between the service access points. Routing rules are described in "Routing" on page 27.

CHAPTER 4

Routing

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

In this chapter

This chapter discusses the following topics:

Artix Routing

Overview

Artix routing is implemented within Artix service access points and is controlled by rules specified in the SAP's contract. Artix SAPs 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, according to some specified criteria. This routing information is all that is required to implement port-based or operation-based routing. Content-based routing requires that application code be written to implement the routing logic.

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 transmits data using SOAP over HTTP, Artix can route the score data to a reporting service that consumes data using CORBA. 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 it 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 11 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 11: *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 12 to the interfaces in Example 11 on page 30, 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 12: *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

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`, the specifies a unique identifier for the route. `route` also has an optional attribute, `multiRoute`, which is discussed in "Advanced Routing Features" on page 39.

<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 29.

`<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.

<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 29.

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 39.

<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 11 on page 30, your Artix contract would contain the elements in Example 13.

Example 13: *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">
       <corba:address location="file://baseball.ref"/>
     </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 13:

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 31.

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 12 on page 31, to getScore defined in finalScorePortType, shown in Example 11 on page 30, your Artix contract would contain the elements in Example 14.

Example 14: *Operation to Operation Routing*

```
<service name="fullScoreService">
  <port binding="tns:fullScoreBinding"
        name="fullScorePort">
    <corba:address="file://score.ref" />
  </port>
</service>
<service name="finalScoreSerice">
  <port binding="tns:finalScoreBinding"
        name="finalScorePort">
    <tuxedo:address serviceName="finalScoreServer" />
  </port>
</service>
<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 14:

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 11 on page 30.The resulting contract would include the fragment shown in Example 15.

Example 15: *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 29.

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.

To do this using the routing editor of the Artix Designer

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 29.

Example 16 shows an Artix contract fragment describing a route for broadcasting a message to a number of ports.

Example 16: *Fanout Broadcasting*

```
<message name="statusAlert">
  <part name="alertType" type="xsd:int"/>
  <part name="alertText" type="xsd:string"/>
</message>
```

Example 16: *Fanout Broadcasting*

```
<portType name="statusGenerator">
  <operation name="eventHappens">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
<portType name="statusChecker">
  <operation name="eventChecker">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
<service name="statusGeneratorService">
  <port binding="tns:statusGeneratorBinding"
        name="statusGeneratorPort">
    <soap:address location="http:\\localhost:8081"/>
  </port>
</service>
<service name="statusCheckerService">
  <port binding="tns:statusCheckerBinding"
        name="statusCheckerPort1">
    <corba:address location="file:\\status1.ref"/>
  </port>
  <port binding="tns:statusCheckerBinding"
        name="statusCheckerPort2">
    <tuxedo:address serviceName="statusService"/>
  </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 in the order that the destinations are listed in the route. 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.

To create a failover route using the Artix Designer...

Given the route shown in Example 17, the message will first be routed to `destinationPortA`. If service on `destinationPortA` cannot receive the message, it is routed to `destinationPortB`.

Example 17: *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:route>
```

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 criteria specified in the listed sub-element.

Each sub-element has a `name` attribute to specify the transport attribute, and most 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 the string is not empty.

For information on the transport attributes for HTTP see "Working with HTTP" on page 179. For information on the transport attributes for WebSphere MQ see "Working with WebSphere MQ" on page 133.

To add transport attributes rules to your route using the Artix Designer...

Example 18 shows a route using transport attribute rules based on HTTP header attributes. Only messages whose `If-Modified-Since` is equal to `"Sat, 29 Oct 1994 19:43:31 GMT"`.

Example 18: *Transport Attribute Rules*

```
<rotuing:route name="httpTransportRoute">
  <routing:source service="tns:httpService"
                  port="tns:httpPort"/>
  <routing:trasnportAttributes>
    <rotuing:equals name="IfModifiedSince"
                    value="Sat, 29 Oct 1994 19:43:31 GMT"/>
  </routing:transportAttributes>
  <routing:destination service="tns:httpDest"
                       port="tns:httpDestPort"/>
</routing:route>
```

Attribute Propagation through Routes

Overview

Often you will need to ensure that message attributes are propagated through the router when it transforms messages between different payload formats or translates it across different transports. Artix can either simply drop the message attributes between the formats or it can use attribute propagation rules specified in the Artix contract describing the system.

The rule describing attribute propagation between two endpoints are specified in the routing section of the Artix contract for the system. Each route must specify the attributes it wants to propagate and for which message it is propagated. If the attribute is not explicitly listed, the router will not propagate it.

Note: There are a few attributes that are included as part of the message body and these are propagated regardless of the specified propagation rules.

Describing attribute propagation rules in an Artix contract

To describe attribute propagation rules in a contract you use two elements. One describes the attributes of the input message passed between the two endpoints. The other describes the attributes of the output message between the two endpoints.

<routing:propagateInputAttribute>

`<routing:propagateInputAttribute>` specifies an attribute from the input message to propagate through the route. It takes one required property, `name`, which specifies the name of the message attribute to be propagated through the route. For example, if you wanted to propagate the attribute `UserName` between two HTTP endpoints you would include the rule shown in Example 19 in your contract's route.

Example 19: *Attribute Propagation Input Rule*

```
<routing:route name="VOD" >
  <routing:propagateInputAttribute name="UserName" />
 ...
</routing:route>
```

`propagateInputAttribute` also takes a second optional property, `target`, that allows you to specify the name of the coressponding attribute name in the destination endpoint's transport. If you do not specify a target, the router assumes that the attribute names for both transports are identical.

For example, if your route is between an HTTP port and a JMS port and you want to propagate the HTTP port's `UserName` attribute to the JMS port's `JMSXUserID` attribute you would include the rule shown in Example 20 in your contract's route.

Example 20: *Attribute Propagation Input Rule with Target*

```
<routing:route name="VOD" >
  <routing:propagateInputAttribute name="UserName"
   target="JMSXUserID" />
 ...
</routing:route>
```

<routing:propagateOutputAttribute>

`<routing:propagateOutputAttribute>` specifies an attribute from the output message to propagate through the route. It takes the same properties as `propagateInputAttributes`.

For example, if you needed the service at the HTTP endpoint in Example 20 needed to validate the UserName of the message returned from the JMS endpoint, you would need to specify that the output message's JMSXUserID was propagated to the HTTP endpoint's UserName attribute by including the rule shown in Example 21 in your contract's route.

Example 21: *Attribute Propagation Output Rule with Target*

```
<routing:route name="VOD" >
  <routing:propagateOutputAttribute name="JMSXUserID"
   target="UserName" />
 ...
</routing:route>
```

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”.

CHAPTER 5

Building Contracts from Java Classes

Artix provides tools for quickly building contracts from Java objects.

Overview

Many applications have been developed using Java to take advantage of Java's platform independence among other things. Java's platform independence is a perfect compliment to Artix's transport independence. To facilitate the integration of Java applications with Artix, Artix provides tools for generating the logical portion of an Artix contract from existing Java classes. These tools use the mapping rules described in Sun's JAX-RPC 1.1 specification.

---

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][-?] 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 XMLSchema information generated to represent the data types inside the WSDL document.By default, the generated target namespace of the XMLSchema 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.

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 5 shows the Java types Artix can map to an Artix contract.

Table 5: *Java to WSDL Mappings*

Java	Artix Contract
`boolean`	`xsd:boolean`

Table 5: *Java to WSDL Mappings*

Java	Artix Contract
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
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 22 shows the mapping for a `java.lang.Float`.

Example 22: *Mapping of java.lang.Float to XMLSchema*

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

Exceptions

Aritx 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>` within the operation representing the corresponding

Java method. The generated <fault> will reference a generated <message> describing the Java exception class. The name attribute of the <message> will be taken from the name of the Java exception class.

Because SOAP only supports <fault> messages with a single <part>, the generated <message> 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 23, you could generate an Artix contract on it by compiling the interface into a .class file and running the command javatowsdl Base.

Example 23: *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 24.

Example 24: *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>
            <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>
```

Example 24: *Base Artix Contract*

```
            <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>
            <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>
```

Example 24: *Base Artix Contract*

```
            <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>
            <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>
```

Example 24: *Base Artix Contract*

```
    <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>
    <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>
```

Example 24: *Base Artix Contract*

```
        <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>
```

CHAPTER 6

Working with CORBA

The CORBA Plug-in allows CORBA applications to be used with an Artix integration solution. It also provides CORBA functionality to Artix applications.

In this chapter

This chapter discusses the following topics:

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

In this section

This section discusses the following topics:

Primitive Type Mapping

Mapping chart

Most primitive IDL types are directly mapped to primitive XML Schema types. Table 6 lists the mappings for the supported IDL primitive types.

Table 6: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
`Any`	`xsd:anyType`	`corba:any`	`IT_Bus::AnyHolder`
`boolean`	`xsd:boolean`	`corba:boolean`	`IT_Bus::Boolean`
`char`	`xsd:byte`	`corba:char`	`IT_Bus::Char`
`wchar`	`xsd:string`	`corba:wchar`	
`double`	`xsd:double`	`corba:double`	`IT_Bus::Double`
`float`	`xsd:float`	`corba:float`	`IT_Bus::Float`
`octet`	`xsd:unsignedByte`	`corba:octet`	`IT_Bus::Octet`
`long`	`xsd:int`	`corba:long`	`IT_Bus::Long`
`long long`	`xsd:long`	`corba:longlong`	`IT_Bus::LongLong`
`short`	`xsd:short`	`corba:short`	`IT_Bus::Short`
`string`	`xsd:string`	`corba:string`	`IT_Bus::String`
`wstring`	`xsd:string`	`corba:wstring`	
`unsigned short`	`xsd:unsignedShort`	`corba:ushort`	`IT_Bus::UShort`
`unsigned long`	`xsd:unsignedInt`	`corba:ulong`	`IT_Bus::ULong`
`unsigned long long`	`xsd:unsignedLong`	`corba:ulonglong`	`IT_Bus::ULongLong`

Unsupported types

Artix does not support the CORBA `long double` type.

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 25.

Example 25: *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 26.

Example 26: *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 27.

Example 27: *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 7 shows the mappings from complex IDL types to XMLSchema, Artix CORBA type, and Artix C++ types.

Table 7: *Complex Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
struct	See Example 28	corba:struct	IT_Bus::SequenceComplexType
enum	See Example 29	corba:enum	IT_Bus::AnySimpleType
fixed	xsd:decimal	corba:fixed	IT_Bus::Decimal
union	See Example 34	corba:union	IT_Bus::ChoiceComplexType
typedef	See Example 37		
array	See Example 39	corba:array	IT_Bus::ArrayT<>
sequence	See Example 45	corba:sequence	IT_Bus::ArrayT<>
exception	See Example 48	corba:exception	IT_Bus::UserFaultException

Structures

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.

For example, the structure defined in Example 2 on page 11, `personalInfo`, can be represented in the CORBA type map as shown in Example 28:

Example 28: *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

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.

For example, the enumeration defined in Example 2 on page 11, hairColorType, can be represented in the CORBA type map as shown in Example 29:

Example 29: *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

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.

For example, the fixed type defined in Example 30, `myFixed`, would be

Example 30: *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 31.

Example 31: *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 32. 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 32: *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

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'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:chioce> is used as the representation of the union throughout the contract.

For example consider the union, myUnion, shown in Example 33:

Example 33: *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 34:

Example 34: *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>
```

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.
`descriminator`	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 33 on page 65, would be described with a CORBA type map entry similar to that shown in Example 35.

Example 35: *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>
```

Example 35: *myUnion CORBA type map*

```
    <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

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>`.

For example, the definition of myLong in Example 36, can be described as

Example 36: *myLong IDL*

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

shown in Example 37:

Example 37: *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

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 38.

Example 38: *myArray IDL*

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

Its logical description will be similar to that shown in Example 39:

Example 39: *myArray logical description*

```
<xsd:complexType name="myArray">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="10" maxOccurs="10" />
  </xsd:sequence>
</xsd:complexType>
```

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 40:

Example 40: *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

Multidimensional arrays are handled by creating multiple arrays and combining them to form the multidimensional array. For example, an array defined as follows:

Example 41: *Multidimensional Array*

```
\\ IDL
typedef long array2d[10][10];
```

generates the following logical description:

Example 42: *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>
```

The corresponding entry in the CORBA type map is:

Example 43: *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

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 44, `longSeq` and `charSeq`:

Example 44: *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 45:

Example 45: *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>
```

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 45 has a CORBA type map description similar to that shown in Example 46:

Example 46: *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

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.

For example, the exception defined in Example 47, `idNotFound`,

Example 47: *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 48:

Example 48: *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 49:

Example 49: *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 XMLSchema allow you define recursive data types. Because both type definition schemes support recursion, Artix directly maps recursive types between IDL and XMLSchema. 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 XMLSchema

Recursive data types are defined in XMLSchema as complex types using the `<complexType>` element. XMLSchema 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 50 shows a recursive complex type XMLSchema type, `allAboutMe`, defined using a named type.

Example 50: *Recursive XMLSchema 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>
```

XMLSchema 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 51, 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 51: *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 XMLSchema types, the IDL generated from the typemap restores the recursion in the IDL type. The IDL generated from the typemap in Example 51 on page 74 defines `allAboutMe` using recursion. Example 52 shows the generated IDL.

Example 52: *IDL for a Recursive Data Type*

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

Mapping XMLSchema Features that are not Native to IDL

Overview

There are a number of data types that you can describe in your Artix contract using XMLSchema 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 type mappings
- Attribute mapping
- Nested choice mapping
- Inheritance mapping
- Nillable mapping

Binary type mappings

There are three binary types defined in XMLSchema 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. For example, the schema type, `joeBinary`, described in Example 53 results in the CORBA typemap description shown in Example 54.

Example 53: *joeBinary schema description*

```
<xsd:element name="joeBinary type="xsd:hexBinary" />
```

The resulting IDL for `joeBinary` is shown in Example 55.

Example 54: *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 55: *joeBinary IDL*

```
\\IDL
typedef sequence<octet> joeBinary;
```

Attribute mapping

Required XMLSchema attributes are treated as normal elements in a CORBA structure.

Note: Attributes are not supported for complex types defined with `<choice>`.

For example, the complex type, `madAttr`, described in Example 56 contains two attributes, `material` and `size`.

Example 56: *madAttr XMLSchema*

```
<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 57. Notice that `size` and `material` are simply incorporated into the `madAttr` structure in the CORBA typemap.

Example 57: *madAttr CORBA typemap*

```
<corba:annonstring bound="3" name="materialType" type="tns:material" />
```

Example 57: *madAttr CORBA typemap*

```
<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 58.

Example 58: *madAttr IDL*

```
\\IDL
struct madAttr
{
  string style;
  char gender;
  long size;
  string<3> material;
}
```

Nested choice 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.

For example, the complex type `details`, shown in Example 59, contains a nested `choice`.

Example 59: *details XMLSchema*

```
<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 60, contains a new union, `detailsChoiceType`, to describe the nested choice. Note that the `type` attribute for both `details` and `detailsChoiceType` have 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 60: *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 61.

Example 61: *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

XMLSchema describes inheritance using the `<complexContent>` tag and the `<extension>` tag. For example the complex type `seaKayak`, described in Example 62, extends the complex type `kayak` by including two new fields.

Example 62: *seaKayak XMLSchema*

```
<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 <compexContent> element. The intermediate type is named by appending an indentifier describing the complex content to the new type's name. Table 8 shows the complex content identifiers used appended to the intermediate type name.

Table 8: *Complex Content Identifiers in CORBA Typemap*

XMLSchema Type	Typemap Identifier
<sequence>	SequenceStruct
<all>	AllStruct
<choice>	ChoiceType

The CORBA type generated to represent the XMLSchema 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 XMLSchema attributes into CORBA types.

Example 63 shows how Artix maps the complex types defined in Example 62 on page 79 into a CORBA type map.

Example 63: *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 62 on page 79 is shown in Example 64.

Example 64: *seaKayak IDL*

```
\\ IDL
struct kayak
{
  long length;
  long width;
  string material;
};
struct seaKayakSequenceStruct
{
  string chines;
  string cockpitStyle;
};
struct seaKayak
{
  kayak kayak_f;
  seaKayakSequenceStruct seqKayakSequenceStruct_f;
};
```

Nillable mapping

XMLSchema 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`.

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 65.

Example 65: *webData XMLSchema*

```
<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 66 shows the CORBA typemap for `webData`.

Example 66: *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 67 shows the IDL for webData.

Example 67: *webData IDL*

```
\\IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
struct webData {
    long zipCode;
    string_nil name;
    string_nil emailAddress;
};
```

Optional attributes

Attributes defined as optional in XMLSchem 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 65 using attributes instead of a sequence. The data description for webData defined with attributes is shown in Example 68.

Example 68: *webData XMLSchema 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>
```

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 69 shows the CORBA typemap for `webData`.

Example 69: *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 70 shows the IDL for `webData`.

Example 70: *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 preplaning in the creation of your XMLSxhema 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 71 shows the syntax for specifying the binding in the type definition.

Example 71: *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 of 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 72 shows an Artix contract fragment that uses Artix references.

Example 72: *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 72: *Reference Sample*

```
  <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>
```

Example 72: *Reference Sample*

```
  <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>
```

Example 72: *Reference Sample*

```
  <portType name="Account">
    <operation name="account_id">
      <input message="tns:account_id" name="account_id"/>
      <output message="tns:account_idResponse"
              name="account_idResponse"/>
    </operation>
    <operation name="balance">
      <input message="tns:balance" name="balance"/>
      <output message="tns:balanceResponse"
              name="balanceResponse"/>
    </operation>
    <operation name="withdraw">
      <input message="tns:withdraw" name="withdraw"/>
      <output message="tns:withdrawResponse"
              name="withdrawResponse"/>
      <fault message="tns:InsufficientFundsException"
   name="InsufficientFunds"/>
    </operation>
    <operation name="deposit">
      <input message="tns:deposit" name="deposit"/>
      <output message="tns:depositResponse"
              name="depositResponse"/>
    </operation>
  </portType>
  <portType name="Bank">
    <operation name="find_account">
      <input message="tns:find_account" name="find_account"/>
      <output message="tns:find_accountResponse"
              name="find_accountResponse"/>
      <fault message="tns:AccountNotFound"
             name="AccountNotFound"/>
    </operation>
    <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 73 shows the generated CORBA typemap resulting from generating both the `Account` and the `Bank` interfaces into the same contract.

Example 73: *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 74 shows the IDL generated for the Bank interface.

Example 74: *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);
};
```

Modifying a Contract to Use CORBA

Overview

Service Access Points (SAPs) that use CORBA require that special binding, port, and type mapping information be added to the physical portion of the Artix contract. The binding definition resolves any ambiguity about parameter order, return values, and type. The port definition specifies the addressing information need by clients or servers to locate the CORBA object. The port can also specify POA policies the exposed CORBA object uses. The type mapping information maps complex schema types, defined in the logical portion of the contract, into CORBA data types.

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"
```

In this section

This section discusses the following topics:

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, maps the parts of a logical message to the proper payload format for CORBA applications. The CORBA binding specifies the repository ID of the IDL interface, resolves parameter order and mode ambiguity, and maps the data types to CORBA data types.

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 `<operation>` 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 member of <corba:operation>. Each <part> 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 s a member 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 member 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.

Using the command line

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] 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

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 the Artix Designer.

Example

For example, the logical operation `personalInfoLookup`, shown in Example 9 on page 19, has a CORBA binding similar to the one shown in Example 75.

Example 75: *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 a 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.

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
  ```

- 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 the *Artix Administration Guide*.

- 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 Application Server Platform 6.0, such as load balancing and fault tolerance, when deploying an Artix integration project. For information on using these advanced CORBA features, see the Application Server Platform 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`*.

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to Example 76:

Example 76: *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 IDL from an Artix Contract

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] 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`.

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 to from which to generate the IDL.

Generating a Contract from IDL

Overview

If you are starting from a CORBA server or client, Artix can build the logical portion of the WSDL contract from IDL. Contracts generated from IDL have CORBA-specific entries and namespaces added.

The IDL 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.

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 compiler ignores any definitions that use unsupported CORBA types. The IDL 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
  {
    A member;
  };

  S get_op();
};
```

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

Using the command line

IONA's IDL compiler supports several command line flags that specify how to create a WSDL file from an IDL file. The IDL compiler is run using the following command:

```
idl -wsdl:[-aaddress][-ffile][-Odir][-turi][-stype][-rfile][-Lfile][-Pfile][-wnamespace]
          [-xnamespace][-tnamespace][-Tfile][-nfile][-b] idlfile
```

The command has the following options:

`-wsdl`	Specifies that WSDL is to be generated. This flag is required.
`-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 contents of this file is incorporated into the WSDL file. The *`file`* must exist when you run the IDL compiler.
`-O`*`dir`*	Specifies the directory into which the WSDL file is written.
`-t`*`uri`*	Specifies the URI for the `corbatm` namespace. This overrides the default.
`-s`*`type`*	Specifies the XMLSchema 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.

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 77.

Example 77: *personalInfoService Interface*

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

Example 77: *personalInfoService Interface*

```
  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 the GUI or the command line. The resulting contract is similar to that shown in Example 78.

Example 78: *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/bindings/corba/typemap">
  <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>
```

Example 78: *personalInfoService Contract*

```
    <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.empId" type="xsd:int"/>
    <xsd:element name="personalInfoService.lookup.return"
 type="xsd1:personalInfoService.personalInfo"/>
    <xsd:element name="personalInfoService.idNotFound"
 type="xsd1:personalInfoService.idNotFound"/>
  </schema>
</types>
<message name="personalInfoService.lookup">
  <part name="empId" element="xsd1:personalInfoService.lookup.empId"/>
</message>
<message name="personalInfoService.lookupResponse">
  <part name="return" element="xsd1:personalInfoService.lookup.return"/>
</message>
<message name="_exception.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:_exception.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>
```

Example 78: *personalInfoService Contract*

```
      <input/>
      <output/>
      <fault name="personalInfoService.idNotFound"/>
    </operation>
  </binding>
  <service name="personalInfoServiceService">
    <port name="personalInfoServicePort" binding="tns:personalInfoServiceBinding">
      <corba:address location="..."/>
    </port>
  </service>
  <corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
    <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>
```

Configuring Artix to Use the CORBA Plug-in

Overview

The CORBA interopability features of Artix are provided through a plug-in. If you are using Artix with the CORBA transport, you need to ensure that the CORBA plug-in is loaded by the Artix runtime and that the plug-in is properly configured.

Loading the plug-in

To configure the Artix runtime to load the CORBA plug-in add `ws_orb` to the `orb_plugins` list for your Artix instance. For example, if your Artix instance is getting its configuration from the configuration scope, the `orb_plugins` list would look like Example 79.

Example 79: *orb_plugin list for CORBA*

```
{
  ...
  corba_interop
  {
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop", "mq", "ws_orb", "fixed"];
     ...
  }
}
```

Plug-in configuration

The CORBA plug-in is configured using the same configuration variables as IONA's Application Server Platform's CORBA implementation. For more information on configuring the CORBA plug-in, see the *Application Server Platform Configuration Reference*.

CHAPTER 7

Working with Tuxedo

Artix easily integrates BEA Tuxedo applications with CORBA and Web service applications.

In this chapter

This chapter discusses the following topics:

Introduction

Overview

Artix provides integration with BEA Tuxedo applications by supporting use of the Tuxedo ATMI transport. Artix also supports Field Manipulation Language (FML) buffers, in Tuxedo Version 7.1 or higher.

Note: BEA Tuxedo integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports BEA Tuxedo integration.

FML support

Artix supports the following FML features:

Table 9: *Artix FML Feature Support*

Feature	Supported	Not Supported
16-bit FML Buffers	x	
32-bit FML Buffers	x	
VIEWS		x
Buffer Pointers		x
Embedded 32-bit FML Buffers		x
Embedded 32-bit Views		x
Character Arrays	x	
Multi-Byte Character Arrays		x
Packed Decimals		x
Multiple Occurrence Fields	x	

Using FML Buffers

Overview

Field Manipulation Language (FML) buffers allow Tuxedo applications to manipulate data stored outside of their application space with ease. FML buffers are described using *field table files* that may be compiled into C header files.

Artix enables non-Tuxedo applications to interact with Tuxedo applications that use FML buffers by translating the data stored in the buffers into data that the non-Tuxedo application can understand. Artix allows the non-Tuxedo application to manipulate the data in the buffer in the same manner as a Tuxedo application.

In this section

This section discusses the following topics:

Mapping FML Buffer Descriptions to Artix Contracts

Overview

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. To create an FML header file from a field table file, you use the Tuxedo `mkfldhdr` and `mkfldhdr32` utility programs.

Mapping to logical type descriptions

Because FML does not provide a means for determining if a field has multiple entries without scanning the buffer, FML buffers must be described as a sequence of sequences. Each field of a buffer is described as an unbounded sequence of the type specified in the field description table. The field elements are ordered in increasing order by their `fldid`.

For example, the `personalInfo` structure, defined in Example 2 on page 11, could be described by the field table file shown in Example 80.

Example 80: *personalInfo Field Table File*

```
# personalInfo Field Table
# name      number     type        flags     comment
name        100        string      -         Person's name
age         102        short       -         Person's age
hairColor   103        string      -         Person's hair color
```

The C++ header file generated by the Tuxedo `mkfldhdr` tool to represent the `personlInfo` FML buffer is shown in Example 81. 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 81: *personalInfo C++ header*

```
/*      fname        fldid           */
/*      -----        -----           */
#define name         ((FLDID)41060)   /* number: 100 type: string */
#define age          ((FLDID)102)     /* number: 102 type: short */
#define hairColor    ((FLDID)41063)   /* number: 103 type: string */
```

The order of the elements in the sequence used to logically describe the FML buffer are ordered in increasing order by `fldid` value. For the `personalInfo` FML buffer age must be listed first in the Artix contract despite the fact that it is the second element listed in the field table. The corresponding logical description of the FML buffer data in an Artix contract is shown in Example 82.

Example 82: *Logical description of personalInfo 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="personalInfoFML16">
      <sequence>
        <element name="age" type="xsd:short" minOccurs="0" maxOccurs="unbounded"/>
        <element name="name" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
        <element name="hairColor" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </schema>
</types>
```

Mapping to the physical FML binding

Artix defines an FML namespace to describe the physical binding of a message to an FML buffer. To include the FML namespace to your Artix contract include the following in the `<definition>` element at the beginning of the contract.

```
xmlns:fml="http://www.iona.com/bus/fml"
```

The FML namespace defines a number of elements to extend the Artix contract's <binding> element. These include:

<fml:binding>

The <fml:binding> element identifies that this binding definition is for an FML buffer. It also specifies the encoding style and transport used with this message.

The encoding style is specified using the mandatory style attribute. The valid encoding styles are doc and rpc.

The transport is specified using the mandatory transport attribute. This attribute can take the URI for any of the valid Artix transport definitions.

<fml:idNameMapping>

The <fml:idNameMapping> element contains the map describing how the element names defined in the logical portion of the contract to the fldid values for the corresponding fields in the FML buffer. This map consists of a series of <fml:element> elements whose fieldName attribute is the name of the logical type describing the element and whose fieldId attribute is the fldid value for the field in the FML buffer. The field elements must be listed in increasing order of their fldid values.

The <fml:idNameMapping> element also specifies if the application is to use FML16 buffers or FML32 buffers. This is done using the mandatory type attribute. type can be either fml16 for specifying FML16 buffers or fml32 for specifying FML32 buffers.

<fml:operation>

The <fml:operation> element is a child of the standard <operation> element. It informs Artix that the operation's messages are to be packed into an FML buffer. <fml:operation> takes a single attribute, name, whose value must be identical to the name attribute of the <operation> element.

Example

For example, the binding for the personalInfo FML buffer, defined in Example 80 on page 112, will be similar to the binding shown in Example 83.

Example 83: *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:xsd1="http://soapinterop.org/xsd"
    xmlns:fml="http://www.iona.com/bus/fml">
...
  <message name="requestInfo">
    <part name="request" type="xsd1:personalInfoFML16"/>
  </message>
  <message name="infoReply">
    <part name="reply" type="xsd1:personalInfoFML16"/>
  </message>

  <portType name="personalInfoPort">
    <operation name="infoRequest">
      <input message="tns:requestInfo" name="requestInfo" />
      <output message="tns:infoReply" name="infoReply" />
    </operation>
  </portType>

  <binding name="personalInfoBinding" type="tns:personalInfoPort">
    <fml:binding style="rpc" transport="http://schemas.iona.com/transports/tuxedo"/>
    <fml:idNameMapping type="fml16">
      <fml:element fieldName="age" fieldId="102" />
      <fml:element fieldName="name" fieldId="41060" />
      <fml:element fieldName="hairColor" fieldId="41063" />
    </fml:idNameMapping>

    <operation name="infoRequest">
      <fml:operation name="infoRequest"/>
      <input name="requestInfo" />
      <output name="infoReply" />
    </operation>
  </binding>
...
</definitions>
```

Using the Tuxedo Transport

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 namespaces:

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

These 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

After defining the Tuxedo services that are endpoints, you must map the operations bound to the port being defined to one of the defined Tuxedo services. This is done using a `<pa:attributeMap>` element. The attribute map element takes one required attribute, `attribute`, that is always set to `serviceName`. The attribute map is defined by `<pa:attributeRule>` elements. Each attribute rule has two attributes:

`value`	Specifies the name of the Tuxedo service on which to invoke.
`operation`	Specifies the operation name from the binding associated with the port being defined.

You must create an attribute rule for all of the operations defined for the port.

Example

An Artix contract exposing the `personalInfoService`, defined in Example 83 on page 115, would contain a `<service>` element similar to Example 84 on page 117.

Example 84: *Tuxedo port description*

```
<service name="personalInfoService">
  <port binding="tns:personalInfoBinding" name="tuxInfoPort">
    <tuxedo:server>
      <tuxedo:service name="personalInfoService" />
    </tuxedo:server>
    <pa:attributeMap attribute="serviceName">
      <pa:attributeRule value="personalInfoService"
                        operation="infoRequest"/>
    </pa:attributeMap>
  </port>
</service>
```

CHAPTER 8

Working with TIBCO Rendezvous

Artix supports the integration of applications using TIBCO Rendezvous and TIBCO JMS messaging systems. Artix also supports the use of the TibrvMsg payload format.

In this chapter

This chapter discusses the following topics:

Introduction

Overview

The TIBCO Rendezvous plug-in lets you use Artix to integrate systems based on TIBCO Rendezvous (TIB/RV) software. TIB/RV uses its own proprietary message schema and transport protocol, and the plug-in bridges these to and from Artix data types, based on a given WSDL contract and the mapping rule. Artix also allows you to send raw XML and opaque data across the TIB/RV messaging transport.

Note: TIBCO Rendezvous integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports TIBCO Rendezvous integration.

Requirements

To use the plug-in, you need to have a TIBCO Rendezvous 7.1 installed on your system. No special configuration is required for running the plug-in. At this time, the plug-in is only supported on Solaris 8 and Windows 2000.

Supported Features

Table 10 shows the matrix of TIBCO Rendezvous features Artix supports.

Table 10: *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

Using TibrvMsg

Overview

Artix supports the use of the TibrvMsg format when using the TIBCO Rendezvous transport.

Binding tags

To use this message format you need to define a binding between the interface you are exposing and the TibrvMsg format. The binding description is placed inside the standard `<binding>` tag and uses the tags listed in Table 11.

Table 11: *TibrvMsg Binding Attributes*

Attribute	Description
`tibrv:binding`	Specifies that the interface is exposed using TibrvMsgs.
`tibrv:binding@stringEncoding`	Specifies the charset used to encode `TIBRVMSG_STRING` data. Use IANA preferred MIME charset names (http://www.iana.org/assignments/character-sets). This parameter must be the same for both client and server.
`tibrv:operation`	Specifies that the operation is exposed using TibrvMsgs.
`tibrv:input`	Specifies that the input message is mapped to a TibrvMsg.
`tibrv:input@sortFields`	Specifies whether the server will sort the input message parts when they are unmarshalled.
`tibrv:input@messageNameFieldPath`	Specifies the field path that includes the input message name.
`tibrv:input@messageNameFieldValue`	Specifies the field value that corresponds to the input message name.
`tibrv:output`	Specifies that the output message is mapped to a TibrvMsg.
`tibrv:output@sortFields`	Specifies whether the client will sort the output message parts when they are unmarshalled.
`tibrv:output@messageNameFieldPath`	Specifies the field path that includes the output message name.

Table 11: *TibrvMsg Binding Attributes*

Attribute	Description
`tibrv:output@messageNameFieldValue`	Specifies the field value that corresponds to the output message name.

TIBRVMSG type mapping

Table 12 shows how TibrvMsg data types are mapped to XSD types in Artix contracts and C++ data types in Artix application code.

Table 12: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD	Artix C++
TIBRVMSG_STRING[1]	xsd:string	IT_BUS::String
TIBRVMSG_BOOL	xsd:boolean	IT_BUS::Boolean
TIBRVMSG_I8	xsd:byte	IT_BUS::Byte
TIBRVMSG_I16	xsd:short	IT_BUS::Short
TIBRVMSG_I32	xsd:int	IT_BUS::Int
TIBRVMSG_I64	xsd:long	IT_BUS::Long
TIBRVMSG_U8	xsd:unsignedByte	IT_BUS::UByte
TIBRVMSG_U16	xsd:unsignedShort	IT_BUS::UShort
TIBRVMSG_U32	xsd:unsignedInt	IT_BUS::UInt
TIBRVMSG_U64	xsd:unsignedLong	IT_BUS::ULong
TIBRVMSG_F32	xsd:float	IT_BUS::Float
TIBRVMSG_F64	xsd:double	IT_BUS::Double
TIBRVMSG_STRING	xsd:decimal	IT_BUS::Decimal
TIBRVMSG_DATETIME[2]	xsd:dateTime	IT_BUS::DateTime
TIBRVMSG_OPAQUE	xsd:base64Binary	IT_BUS::Base64Binary
TIBRVMSG_OPAQUE	xsd:hexBinary	IT_BUS::HexBinary
TIBRVMSG_MSG[3]	xsd:complexType/sequence	IT_BUS::SequenceComplexType

Table 12: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD	Artix C++
TIBRVMSG_MSG[4]	xsd:complexType/all	IT_BUS::AllComplexType
TIBRVMSG_MSG[5]	xsd:complexType/choice	IT_BUS::ChoiceComplexType
TIBRVMSG_*ARRAY/MSG[6]	xsd:complexType/sequence with element MaxOccurs > 1	IT_BUS::Array
TIBRVMSG_*ARRAY/MSG[6]	SOAP-ENC:Array[7]	IT_BUS::Array
TIBRVMSG_MSG[3]	SOAP-ENV:Fault[8]	IT_BUS::FaultException

1. TIB/RV does not provide any mechanism to indicate the encoding of strings in a TibrvMsg. The TIBCO plug-in port definition includes a property, `stringEncoding`, for specifying the string encoding. However, neither TIB/RV nor Artix look at this attribute; they merely pass the data along. It is up to the application developer to handle the encoding details if desired.
2. `TIBRVMSG_DATATIME` has microsecond precision. However, `xsd:dateTime` has only millisecond precision. Therefore, when using Artix sub-millisecond percision will be lost.
3. Sequences are mapped to nested messages where each element is a separate field. These fields are placed in the same order as they appear in the original sequence with field IDs beginning at 1. The fields are accessed by their field ID.
4. Alls are mapped to nested messages where each elements is mapped to a separate field. The fields representing the elements of the all are given the same field name as element name and field IDs beginning from 1. They can be accessed by field name beginning from field ID 1. That means that the order of fields can be changed.
5. Choices are mapped to nested messages where each elements is a separate field. Each field is enclosed with the same field name/type as element name/type of active member, and accessed by field name with field ID 1.
6. Arrays having `integer` or `float` elements are mapped to appropriate TIB/RV array types; otherwise they are mapped to nested messages.

7. SOAP RPC-encoded multi-dimensional arrays will be treated as one-dimensional: e.g. a 3x5 array will be serialized as a one-dimensional array having 15 elements. To keep dimensional information, use nested sequences with `maxOccurs > 1` instead.
8. When a server response message has a fault, it includes a field of type `TIBRVMSG_MSG` with the field name `fault` and field ID 1. This submessage has two fields of `TIBRVMSG_STRING`. One is named `faultcode` and has field ID 1, and the other is named `faultstring` and has field ID 2.

Using the TIB/RV Transport

Overview

Artix contract descriptions of TIB/RV ports use a number of Artix specific WSDL extensions. These extensions allow you to specify a number of TIB/RV properties for the port.

In this section

This section discusses the following topics:

Understanding the TIB/RV Port Properties

Port attributes

Table 13 lists the Artix contract elements used to describe a TIB/RV port.

Table 13: *TIB/RV Transport Properties*

Attribute	Explanation
`tibrv:port`	Indicates that the port uses the TIB/RV transport.
`tibrv:port@serverSubject`	A required element that specifies the subject to which the server listens. This parameter must be the same between client and server.
`tibrv:port@clientSubject`	Specifies the subject that the client listens to. The default is to use the transport inbox name. This parameter only affects clients.
`tibrv:port@bindingType`	Specifies the message binding type.
`tibrv:port@callbackLevel`	Specifies the server-side callback level when TIB/RV system advisory messages are received.
`tibrv:port@responseDispatchTimeout`	Specifies the client-side response receive dispatch timeout.
`tibrv:port@transportService`	Specifies the UDP service name or port for TibrvNetTransport.
`tibrv:port@transportNetwork`	Specifies the binding network addresses for TibrvNetTransport.
`tibrv:port@transportDaemon`	Specifies the TCP daemon port for the TibrvNetTransport.
`tibrv:port@transportBatchMode`	Specifies if the TIB/RV transport uses batch mode to send messages.
`tibrv:port@cmSupport`	Specifies if Certified Message Delivery support is enabled.
`tibrv:port@cmTransportServerName`	Specifies the server's TibrvCmTransport correspondent name.

Table 13: *TIB/RV Transport Properties*

Attribute	Explanation
`tibrv:port@cmTransportClientName`	Specifies the client TibrvCmTransport correspondent name.
`tibrv:port@cmTransportRequestOld`	Specifies if the endpoint can request old messages on start-up.
`tibrv:port@cmTransportLedgerName`	Specifies the TibrvCmTransport ledger file.
`tibrv:port@cmTransportSyncLedger`	Specifies if the endpoint uses a synchronous ledger.
`tibrv:port@cmTransportRelayAgent`	Specifies the endpoint's TibrvCmTransport relay agent.
`tibrv:port@cmTransportDefaultTimeLimit`	Specifies the default time limit for a Certified Message to be delivered.
`tibrv:port@cmListenerCancelAgreements`	Specifies if Certified Message agreements are canceled when the endpoint disconnects.
`tibrv:port@cmQueueTransportServerName`	Specifies the server's TibrvCmQueueTransport correspondent name.
`tibrv:port@cmQueueTransportClientName`	Specifies the client's TibrvCmQueueTransport correspondent name.
`tibrv:port@cmQueueTransportWorkerWeight`	Specifies the endpoint's TibrvCmQueueTransport `worker weight`.
`tibrv:port@cmQueueTransportWorkerTasks`	Specifies the endpoint's TibrvCmQueueTransport `worker tasks` parameter.
`tibrv:port@cmQueueTransportSchedulerWeight`	Specifies the TibrvCmQueueTransport `scheduler weight` parameter.
`tibrv:port@cmQueueTransportSchedulerHeartbeat`	Specifies the endpoint's TibrvCmQueueTransport `scheduler heartbeat` parameter.
`tibrv:port@cmQueueTransportSchedulerActivation`	Specifies the TibrvCmQueueTransport `scheduler activation` parameter.
`tibrv:port@cmQueueTransportCompleteTime`	Specifies the TibrvCmQueueTransport `complete time` parameter.

tibrv:port@bindingType

`tibrv:port@bindingType` specifies the message binding type. TIB/RV Artix ports support three types of payload formats as described in Table 14.

Table 14: *TIB/RV Supported Payload formats*

Setting	Payload Formats	TIB/RV Message Implications
`msg`	TibrvMsg	The top-level messages will have fields of type `TIBRVMSG_STRING`. The value of each field is the name of a WSDL part name from the corresponding WSDL message. If the WSDL part is a primitive type then the value of this type is put against the name of the WSDL part. If the WSDL part is a complex type then a nested TibrvMsg is created and added against the WSDL part name.
`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`.

tibrv:port@callbackLevel

`tibrv:port@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.

tibrv:port@responseDispatchTimeout

`tibrv:port@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.

tibrv:port@transportService

`tibrv:port@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.

tibrv:port@transportNetwork

`tibrv:port@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.

tibrv:port@transportDaemon

`tibrv:port@transportDaemon` specifies the TCP daemon port for TibrvNetTransport. The default is to use `7500` for the `TRDP` daemon, or `7550` for the `PGM` daemon.

tibrv:port@transportBatchMode

`tibrv:port@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.

tibrv:port@cmSupport

`tibrv:port@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.

tibrv:port@cmTransportServerName

`tibrv:port@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.

tibrv:port@cmTransportClientName

`tibrv:port@cmTransportClientName` specifes the client's TibrvCMTransport correspondent name. The default is to use a transient correspondent name.

tibrv:port@cmTransportRequestOld	`tibrv:port@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.
tibrv:port@cmTransportLedgerName	`tibrv:port@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.
tibrv:port@cmTransportSyncLedger	`tibrv:port@cmTransportSyncLedger` Specifies if the endpoint uses a synchronous ledger. `true` specifies that the endpoint uses a synchronous ledger. The default is `false`.
tibrv:port@cmTransportRelayAgent	`tibrv:port@cmTransportRelayAgent` Specifies the endpoint's TibrvCmTransport relay agent. If this property is not set, the endpoint does not use a relay agent.
tibrv:port@cmTransportDefaultTimeLimit	`tibrv:port@cmTransportDefaultTimeLimit` specifies TibrvCmTransport message default time limit. The default is that no message time limit will be set.
tibrv:port@cmListenerCancelAgreements	`tibrv:port@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`.
tibrv:port@cmQueueTransportServerName	`tibrv:port@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.

tibrv:port@cmQueueTransportClientName

`tibrv:port@cmQueueTransportClientName` specifies the client's TibrvCmQueueTransport correspondent name. If this property is set, the client listener joins to the distributed queue of the specifies name. This parameter must be the same among all client queue members.

Note: If distributed queue is enabled on the client side, the transport does not handle any request-response semantics. This is for load-balanced polling-type clients, e.g. one client in the distributed queue periodically invokes an operation that only has outputs and no input, and one listener in the group processes the response.

tibrv:port@cmQueueTransportWorkerWeight

`tibrv:port@cmQueueTransportWorkerWeight` specifies the endpoint's TibrvCmQueueTransport `worker weight`. The default is `TIBRVCM_DEFAULT_WORKER_WEIGHT`.

tibrv:port@cmQueueTransportWorkerTasks

`tibrv:port@cmQueueTransportWorkerTasks` specifies the endpoint's TibrvCmQueueTransport `worker tasks` parameter. The default is `TIBRVCM_DEFAULT_WORKER_TASKS`.

tibrv:port@cmQueueTransportSchedulerWeight

`tibrv:port@cmQueueTransportSchedulerWeight` specifies the TibrvCmQueueTransport `scheduler weight` parameter. The default is `TIBRVCM_DEFAULT_SCHEDULER_WEIGHT`.

tibrv:port@cmQueueTransportSchedulerHeartbeat

`tibrv:port@cmQueueTransportSchedulerHeartbeat` specifies the TibrvCmQueueTransport `scheduler heartbeat` parameter. The default is `TIBRVCM_DEFAULT_SCHEDULER_HB`.

tibrv:port@cmQueueTransportSchedulerActivation

`tibrv:port@cmQueueTransportSchedulerActivation` Specifies the TibrvCmQueueTransport `scheduler activation` parameter. The default is `TIBRVCM_DEFAULT_SCHEDULER_ACTIVE`.

tibrv:port@cmQueueTransportCompleteTime

`tibrv:port@cmQueueTransportCompleteTime` specifies the TibrvCmQueueTransport `complete time` parameter. The default is `0`.

Adding a TIB/RV Port to an Artix Contract

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.

Example

Example 85 shows an Artix description for a TIB/RV port.

Example 85: *TIB/RV Port Description*

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

CHAPTER 9

Working with WebSphere MQ

Artix provides the ability to integrate with IBM WebSphere MQ applications or provide WebSphere MQ qualities of service to non-WebSphere MQ applications.

In this chapter

This chapter discusses the following topics:

Introduction

Overview

Artix provides connectivity to IBM's WebSphere MQ messaging system. This connectivity opens several opportunities for using Artix. The most obvious use is to integrate non-WebSphere MQ applications with WebSphere MQ applications. Another powerful use of Artix's WebSphere MQ connectivity is writing Artix code that leverages WebSphere MQ qualities of service to provide enterprise class solutions.

Note: IBM WebSphere MQ integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports IBM WebSphere MQ integration.

Integration with synchronous messaging models

Because Artix abstracts the details of the messaging infrastructure from the application level code, Artix allows for a seamless integration between WebSphere MQ, which uses an asynchronous messaging model, and applications that use a synchronous messaging model. Asynchronous WebSphere MQ applications will still send messages without blocking and poll the reply queue for a response if one is expected. Synchronous applications, such as CORBA applications, will continue to block between making a request and receiving a response. Neither end needs to be aware of how the other end handles messages.

Supported Features

Table 15 shows the matrix of WebSphere MQ features Artix supports.

Table 15: *Supported WebSphere MQ Features*

Feature	Supported	Not Supported
Dynamic Queue Creation	x	
SSL	x	
Queue Manager Clustering		x
LDAP		x

Table 15: *Supported WebSphere MQ Features*

Feature	Supported	Not Supported
Channel Process Pooling		x
Wildcards for Security Settings		x

Describing an Artix WebSphere MQ Port

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 proprietary WSDL extensions to specify all of the attributes that can be set on an WebSphere MQ port. The XMLSchema describing the extensions used for the WebSphere MQ port definition is included in the Artix installation under the `schemas` directory.

In this section

This section discusses the following topics:

Configuring an Artix WebSphere MQ Port

Overview

The Artix WebSphere MQ port description distinguishes between ports used for server applications and ports used by client applications because the port attributes have different implications for serer application and client applications. Many of the attributes that can be set in an MQ message descriptor are definable using attributes to the MQ port definition.

WebSphere MQ namespace

The WSDL extensions used to describe WebSphere MQ transport details are defined in the WSDL namespace `http://schemas.iona.com/bindings/mq`. To use the WebSphere MQ extensions you will need to include the following in the `<definitions>` tag of your contract:

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

WebSphere MQ port elements

When describing an WebSphere MQ port in your Artix contract you use two child elements to the port:

<mq:client> defines a port for a WebSphere MQ client application.

<mq:server> defines a port a WebSphere MQ server application.

You must use at least one of these elements in your Artix WebSphere MQ port description.

WebSphere MQ port attributes

Table 16 lists the attributes that are use to define the properties of a WebSphere MQ port. They are described in detail in the section that follow the table.

Table 16: *WebSphere MQ Port Attributes*

Attributes	Description
`QueueManager`	Specifies the name of the queue manager.
`QueueName`	Specifies the name of the message queue.

Table 16: *WebSphere MQ Port Attributes*

Attributes	Description
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.
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.
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 decscriptor's MsgId field..

Table 16: *WebSphere MQ Port Attributes*

Attributes	Description
`CorrelationId`	Specifies the value for the MQ message decscriptor'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 in the messages in the queue need to be converted to the system's native encoding.
`ApplicationIdData`	Specifies the value for the MQ message decscriptor's `ApplIdentityData` field.
`ApplicationOriginData`	Specifies the value for the MQ message decscriptor's `ApplOriginData` field.
`UserIdentification`	Specifies the value for the MQ message decscriptor's `UserIdentifier` field.

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 86 shows a simple WebSphere MQ server port configuration for servers that listen for requests using a queue manager called `leo`.

Example 86: *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 87 shows a definition of a simple WebSphere MQ client that places oneway requests onto a queue called `ether`.

Example 87: *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 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 `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 88 shows a WebSphere MQ server port that defaults to placing reply messages onto the queue `outbox`.

Example 88: *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 `ReplyToQMgr` 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 89 shows a WebSphere MQ client port that is configured to receive replies from the server defined in Example 88 on page 143.

Example 89: *MQ Client with ReplyQueueName Set*

```
<mq:client QueueName="ether" QueueManager="leo"
           ReplyQueueName="outbox" ReplyQueueManager="pager" />
```

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 recieve 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` cuases 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 3, 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.

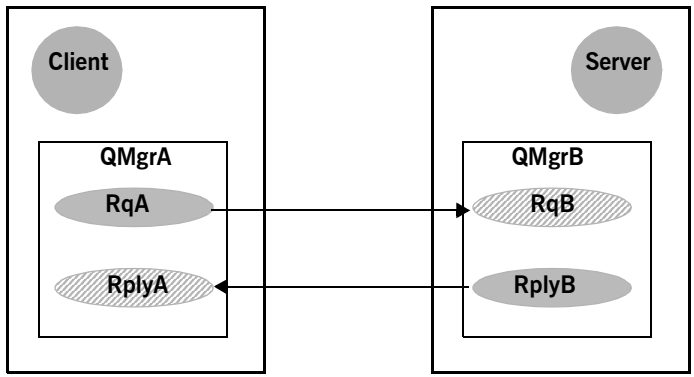

Figure 3: *MQ Remote Queues*

The Artix WebSphere MQ port definitions for the client and server for this deployment are shown in Example 90. AliasQueueName is set to RplyB because that is the remote queue proxying for the reply queue on 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 asured that the client is populating the request's message descriptor with the needed information for the server to determine where replies are sent,

Example 90: *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 91 shows a WebSphere MQ client port that loads its request queue manager in process.

Example 91: *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 17.

Table 17: *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 92, 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 92: *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 18 shows the actions of `MQGET` and `MQPUT` when receiving a message using a WSDL specified message ID and a WSDL specified correlation ID.

Table 18: *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 93 shows a WebSphere MQ client application that wants to correlate messages using the `messageID copy` setting.

Example 93: *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, recive, receive exclusive, and receive shared. Each setting mapping corresponds to a WebSphere MQ setting for the MQOPEN. The default is recieve.

Attribute settings

Table 19 describes the correlation between the Artix attribute settings and the MQOPEN settings.

Table 19: *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 it. 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).
recieve 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 19: *Artix WebSphere MQ Access Modes*

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 94 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 94: *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 cooresponding 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 95 shows the settings for a MQ client port where replies are required in at most 3 minutes.

Example 95: *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 96 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 96: *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 97 shows the settings for a WebSphere MQ port that sends persistent oneway messages.

Example 97: *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 this attribute are explained in Table 20.

Table 20: *Transactional Attribute Settings*

Attribute Setting	Description
`none (Default)`	The messages are not part of a transaction. No roleback 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 transaction with WebSphere MQ serving as the resource manager.

Example

Example 98 shows the settings for a WebSphere MQ client 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 98: *MQ Client setup to use Transactions*

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

ReportOption

Overview

`ReportOption` is mapped 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 21.

Table 21: *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 21: *ReportOption Attribute Settings*

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

Example

Example 99 shows the settings for a WebSphere MQ client that wants to be notified if any of its message expire before they are delivered.

Example 99: *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 22.

Table 22: *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 22: *FormatType Attribute Settings*

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 100 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 100: *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 101 shows the settings for a WebSphere MQ client that wants to use message Ids to correlate response and request messages.

Example 101: *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 102 shows the settings for a WebSphere MQ client that wants to use correlation Ids to correlate response and request messages.

Example 102: *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 103 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 103: *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 systems 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 104 shows a WebSphere MQ client port used for making requests against a server on a Unix system.

Example 104: *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 105 shows the settings for a WebSphere MQ client that wants to identify itself to the server.

Example 105: *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 106 shows the settings for a WebSphere MQ client that needs to specify the User that is making the request.

Example 106: *WebSphere MQ Client Sending UserID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           UserIdentification="tux" />
```

Adding an WebSphere MQ Port to an Artix Contract

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/bindings/mq`. To use the WebSphere MQ extensions you will need to include the following in the `<definitions>` tag of your contract:

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

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 107.

Example 107: *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>
```

CHAPTER 10

Working with the Java Messaging System

Artix allows C++ applications to take advantage of the Java Messaging System.

Overview

The Java Messaging System(JMS) provides a standardized means for Java applications to send messages. Artix provides a transport plug-in that enables systems to place and receive messages from JMS message queues and topics. One large advantage of this is that Artix allows C++ applications to interact directly with Java applications over JMS.

Note: JMS integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports JMS integration.

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. The destinations are specified in the Artix contract describing the application and can be changed without any change in the application code. 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 Artix JMS transport supports the following Artix payload format bindings:

- SOAP
- Fixed
- Tagged
- XML

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, 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

Artix JMS ports are configured entirely in the Artix contract describing your service. The JMS port configuration is done by using a `<jms:address>` element in your service's `<port>` description. `<jms:address>` takes six required attributes to configure the JMS connection:

`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 InitalContextFactory settings" on page 177.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which Artix connects.
`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`.

Example

Example 108 shows an example of an Artix JMS port specification.

Example 108: *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 InitalContextFactory settings

The usual method of specifying the JNDI is to enter the class name provided by your JNDI provider. In Example 108, 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 109, the JNDI service would instantiate a context factory with the class name `com.iona.jbus.jms.naming.sonic.sonicURLContextFactory` to perform lookups.

Example 109: *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 109, 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.

CHAPTER 11

Working with HTTP

The HTTP plug-in lets you configure an Artix integration solution to use the HTTP transport. This chapter first provides a brief introductory overview of HTTP. It then provides a description of the WSDL extensions involved. Finally it provides an overview of the WSDL extension schema that supports the use of HTTP with Artix.

In this chapter

This chapter discusses the following topics:

HTTP Overview

Overview

This section provides an introductory overview of the hypertext transport protocol (HTTP). The following topics are discussed:

Note: A complete introduction to HTTP is outside the scope of this guide. For more details about HTTP see the W3C HTTP specification at `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

What is HTTP?

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

HTTP is termed an *application protocol*. It defines how messages between web browsers and web servers should be formatted and transmitted. It also defines how web browsers and web servers should behave in response to various commands.

Resources and URLs

The files and other information that can be transmitted are collectively known as *resources*. A resource is basically a block of information. Files are the most common example of resources and they can be in various multimedia formats, such as text, graphics, sound, and video. Other examples of resources are server-side script output or dynamically generated query results.

A resource is identifiable by a uniform resource locator (URL). As its name suggests, a URL is the address or location of a resource. A URL typically consists of protocol information followed by host (and optionally port) information followed by the full path to the resource. HTTP is not the only protocol or mechanism for data transfer; other examples include TELNET or the file transfer protocol (FTP). Each of the following is an example of a URL:

- `http://www.iona.com/support/docs/index.xml`
- `ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf`
- `telnet://xyz.com`

In the first of the preceding examples, `http:` denotes that the protocol for data transfer is HTTP, `//www.iona.com` denotes the hostname where the resource resides, and `/support/docs/index.xml` is the full path to the resource (in this case, an XML text file). The other URLs follow similar patterns.

HTTP transaction processing

When a web user on the client-side requests a resource, either by typing a URL or by clicking on a hypertext link, the client browser builds an HTTP request and opens a TCP/IP socket connection to send the request to the internet protocol (IP) address for the host denoted by the URL for the requested resource. The web server host contains an HTTP daemon that waits for client browser requests and handles them when they arrive. When the HTTP daemon receives a request, the requested resource is then returned to the client browser. The server's response can take the form of HTML pages and possibly other programs in the form of ActiveX controls or Java applets.

Format of HTTP client requests

The following is an example of the typical format of an HTTP client request:

```
GET REQUEST-URI HTTP/1.1
header field: value
header field: value

HTTP request body (if applicable)
```

The preceding code can be explained as follows:

`GET`

This is an HTTP method that instructs the server to return the requested resource.

Other HTTP methods might be used here instead. These include:

- `HEAD`—this instructs the server to just return information about the resource (in headers) but not the actual resource itself.
- `POST`—this can be used if you want to send data in the body of the request for subsequent processing by the server.
- `PUT`—this can be used to replace the contents of the target resource with data from the client.

Note: `GET` is the most commonly used method in HTTP client requests.

`REQUEST-URI`

This represents the URL of the resource that the client is requesting. The typical format of a URL is:

`http://`*`hostname/path-to-resource`*

For example:

`http://www.iona.com/support/docs/index.xml`

`HTTP/1.1`

This indicates that the client is using HTTP to transmit the request, and the version of HTTP that the client is using (in this example, `1.1`).

`header field`

Header information can be included to provide information about the request. In HTTP 1.1, the only mandatory header field is `Host:`, to identify the host where the requested resource resides.

In Artix, a number of HTTP client request headers can be configured and sent as part of a client request to a server. See "HTTP WSDL Extensions" on page 187 and "Server Transport Attributes" on page 210 for more details.

`HTTP request body`

This can contain user-entered data or files that are being sent to the server for processing.

Note: This is typically blank in an HTTP request unless the `PUT` or `POST` method is specified.

Format of HTTP server responses

The following is an example of the typical format of an HTTP server response:

```
HTTP/1.1 200 OK
header field: value
header field: value

HTTP response body
```

The preceding code can be explained as follows:

`HTTP/1.1` This indicates that the server is using HTTP to transmit the response, and the version of HTTP that the server is using (in this example, `1.1`).

`200 OK` This is status information that indicates whether the request was processed successfully. The 3-digit code is meant to be machine-readable, and the accompanying descriptive text is for human consumption.

Status codes can be broadly described as follows:

- 2*xx*—A status code starting with `2` means the request was processed successfully.
- 3*xx*—A status code starting with `3` means the resource is now located elsewhere and the client should redirect the request to that new location.
- 4*xx*—A status code starting with `4` means that the request has failed because the client has either sent a request in the wrong syntax, or it might have requested a resource that is invalid or that it is not authorized to access.
- 5*xx*—A status code starting with `5` means that the request has failed because the server has experienced internal problems or it does not support the request method specified.

header field	Header information can be included to provide information about the response itself or about the information contained in the body of the response. In Artix, a number of HTTP server response headers can be configured and sent as part of the server response to the client. See "HTTP WSDL Extensions" on page 187 and "Client Transport Attributes" on page 212 for more details.
HTTP response body	This is where the requested resource is returned to the client, if the request has been processed successfully. Otherwise, it might contain some explanatory text as to why the request was not processed successfully. The data in the body of the response can be in a variety of formats, such as HTML or XML text, GIF or JPEG image, and so on.

HTTP properties

The basic properties of HTTP can be summarized as follows:

- Comprehensive addressing—The target resource on which a client request is to be invoked is indicated by means of a universal resource identifier (URI), either as a location (URL) or name (URN). As explained in "Resources and URLs" on page 180, a URL consists of protocol information followed, typically, by host (and optionally port) information followed by the full path to the resource. For example:

  ```
  http://www.iona.com/support/docs/index.xml
  ```

  See "Resources and URLs" on page 180 for more details.
- Request/response paradigm—A client (web browser) can establish an HTTP connection with a web server by means of a URI, to send a request to that server. See "Format of HTTP client requests" on page 181 for details of the format of a client request message. See "Format of HTTP server responses" on page 183 for details of the format of a server response message.
- Connectionless protocol—HTTP is termed a connectionless protocol because an HTTP connection is typically closed after a single request/response operation. While it is possible for a client to request the server to keep a connection open for subsequent request/response

operations, the server is not obliged to keep the connection open. The advantage of closing connections is that it does not incur any overhead in terms of session housekeeping; however, the disadvantage is that it makes it difficult to track user behavior.

Note: A potential workaround to tracking user behavior is through the use of cookies. A cookie is a string sent by a web server to a web browser and which is then sent back to the web server again each time the browser subsequently contacts that server.

- Stateless protocol—Because HTTP connections are typically closed after each request/response operation, there is no memory or footprint between connections. A workaround to this, in CGI applications, is to encode state information in hidden fields, in the path information, or in URLs in the form returned to the client browser. State can also be saved in a file, rather than being encoded, as in the typical example of a visitor counter program, where state is identified by means of a unique identifier in the form of a sequential integer.
- Multimedia support—HTTP supports the transfer of various types of data, such as text (for example, HTML or XML files), graphics (for example, GIF or JPEG files), sound, and video. These types are commonly referred to as multipart internet mail extension (MIME) types. A server response can include header information that informs the client of the MIME type of the information being sent by the server.
- Proxies and caches—The communication chain between a client and server might include intermediary programs known as proxies. A proxy can receive client requests, possibly modify the request in some way, and then forward the request along the chain possibly to another proxy or to the target server. Such intermediaries can employ caches to store responses that might be appropriate for subsequent requests. Caches can be shared (public) or private. Specific directives can be established in relation to cache behavior and not all responses might be cacheable.

- Security—Secure HTTP connections that run over the secure sockets layer (SSL) or transport layer security (TLS) protocol can also be established. A secure HTTP connection is referred to as HTTPS and uses port 443 by default. (A non-secure HTTP connection uses port 80 by default.)

  Note: See "HTTP WSDL Extensions" on page 187 for details of the various SSL-related configuration attributes that can be used in extending a WSDL contract.

HTTP WSDL Extensions

Overview

This section provides an overview and description of the attributes that you can configure as extensions to a WSDL contract for the purposes of using the HTTP transport plug-in with Artix.

In this section

This section discusses the following topics:

HTTP WSDL Extensions Overview

Overview

This subsection provides an overview of the WSDL extensions involved in configuring the HTTP transport plug-in for use with Artix.

Configuration layout

Example 110 shows (in bold) the WSDL extensions used to configure the HTTP transport plug-in for use with Artix. (Ellipses (that is, …) are used to denotes sections of the WSDL that have been omitted for brevity.)

Example 110: *HTTP configuration WSDL extensions*

```
<definitions…
xmlns:http=”http://schemas.iona.com/transports/http"
xmlns:http-conf="http://schemas.iona.com/transports/http/configu
   ration"
…
<service name="…">
    <port binding="…">
        <http-conf:client SendTimeout="…"
                          ReceiveTimeout="…"
                          AutoRedirect="…"
                          UserName="…"
                          Password="…"
                          AuthorizationType="…"
                          Authorization="…"
                          Accept="…"
                          AcceptLanguage="…"
                          AcceptEncoding="…"
                          ContentType="…"
                          Host="…"
                          Connection="…"
                          ConnectionAttempts="..."
                          CacheControl="…"
                          Cookie="…"
                          BrowserType="…"
                          Referer="…"
                          ProxyServer="…"
                          ProxyUserName="…"
                          ProxyPassword="…"
                          ProxyAuthorizationType="…"
                          ProxyAuthorization="…"
                          UseSecureSockets="…"
```

Example 110:*HTTP configuration WSDL extensions*

```
                        ClientCertificate="…"
                        ClientCertificateChain="…"
                        ClientPrivateKey="…"
                        ClientPrivateKeyPassword="…"
                        TrustedRootCertificate="…"/>

      <http-conf:server SendTimeout="…"
                        ReceiveTimeout="…"
                        SuppressClientSendErrors="…"
                        SuppressClientReceiveErrors="…"
                        HonorKeepAlive="…"
                        RedirectURL="…"
                        CacheControl="…"
                        ContentLocation="…"
                        ContentType="…"
                        ContentEncoding="…"
                        ServerType="…"
                        UseSecureSockets="…"
                        ServerCertificate="…"
                        ServerCertificateChain="…"
                        ServerPrivateKey="…"
                        ServerPrivateKeyPassword="…"
                        TrustedRootCertificate="…"/>
```

HTTP WSDL Extensions Details

Overview

This subsection describes each of the configuration attributes that can be set up as part of the WSDL extensions for configuring the HTTP transport plug-in for use with Artix. It discusses the following topics:

- "Server configuration attributes" on page 190.
- "Client configuration attributes" on page 197.

Server configuration attributes

Table 23 describes the server-side configuration attributes for the HTTP transport that are defined within the `http-conf:server` element.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`SendTimeout`	This specifes the length of time, in milliseconds, that the server can continue to try to send a response to the client before the connection is timed out. The timeout value is at the user's discretion. The default is `30000`.
`ReceiveTimeout`	This specifies the length of time, in milliseconds, that the server can continue to try to receive a request from the client before the connection is timed out. The timeout value is at the user's discretion. The default is `30000`.
`SuppressClientSendErrors`	This 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`	This 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.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`HonorKeepAlive`	This 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. Valid values are `true` and `false`. The default is `false`, to close the connection after a server response is sent. If set to `true`, 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 (via the client-side `Connection` configuration attribute). Otherwise, the connection is closed. If set to `false`, the socket is automatically closed after a server response is sent, even if the client has requested the server to keep the connection alive (via the client-side `Connection` configuration attribute).
`RedirectURL`	This 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.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`CacheControl`	This 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. Valid values are: • `no-cache`—This prevents a cache from using 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`—This indicates that a response can be cached by any cache. • `private`—This indicates that a response is intended only for a single user and cannot be cached by a public (*shared*) cache. 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`—This indicates that a cache must not store any part of a response or any part of the request that evoked it. • `no-transform`—This indicates that a cache must not modify the media type or location of the content in a response between a server and a client. • `must-revalidate`—This indicates that if a cache entry relates to a server response that has exceeded its expiration time, the cache must revalidate that cache entry with the server before it can be used in a subsequent response. • `proxy-revalidate`—This indicates 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.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
	• `max-age`—This indicates that the client can accept a response whose age is no greater than the specified time in seconds. • `s-maxage`—This indicates the same as `max-age`, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by `s-maxage` overrides the age specified by `max-age`. If using this directive, the `proxy-revalidate` directive must also be used. • `cache-extension`—This indicates additional extensions to the other cache directives. Extensions might be informational (that is, do not require a change in cache behavior) or behavioral (that is, act as modifiers to the existing base of cache directives). 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. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.
`ContentLocation`	This specifies the URL where the resource being sent in a server response is located. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`ContentType`	This specifies the media type of the information being sent in a server response (for example, text/html, image/gif, and so on). This is also known as the multipurpose internet mail extensions (MIME) type. 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`. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.
`ContentEncoding`	This 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.
`ServerType`	This specifies what type of server is sending the response to the client. Values in this case take the form *`program-name/version`*. For example, `Apache/1.2.5`. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`UseSecureSockets`	This indicates whether the server wants a secure HTTP connection running over 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 server 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.
`ServerCertificate`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This specifies the full path to the PEM-encoded X509 certificate issued by the certificate authority for the server. For example: `c:\aspen\x509\certs\key.cert.pem` A server must present such a certificate, so that the client can authenticate the server.
`ServerCertificateChain`	This is only relevant if the HTTP connection is running securely over SSL or TLS. PEM-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 PEM-encoded X509 certificates to be presented to the client for verification. This specifies the full path to the file that contains all the certificates in the chain. For example: `c:\aspen\x509\certs\key.cert.pem`
`ServerPrivateKey`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This is used in conjuction with `ServerCertificate`. It specifies the full path to the PEM-encoded private key that corresponds to the X509 certificate specified by `ServerCertificate`. For example: `c:\aspen\x509\certs\privkey.pem` This is required if, and only if, `ServerCertificate` has been specified.

Table 23: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`ServerPrivateKeyPassword`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This specifies a password that is used to decrypt the PEM-encoded private key, if it has been encrypted with a password. The certificate authority typically encrypts these keys when sending them over a public network, and the password is delivered by a secure means.
`TrustedRootCertificate`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This specifies the full path to the PEM-encoded X509 certificate for the certificate authority. For example: `c:\aspen\x509\ca\cacert.pem` This is used to validate the certificate presented by the client.

Client configuration attributes

Table 24 describes the client-side configuration attributes for the HTTP transport that are defined within the `http-conf:client` element.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`SendTimeout`	This specifies the length of time, in milliseconds, that the client can continue to try to send a request to the server before the connection is timed out. The timeout value is at the user's discretion. The default is `30000`.
`ReceiveTimeout`	This specifies the length of time, in milliseconds, that the client can continue to try to receive a response from the server before the connection is timed out. The timeout value is at the user's discretion. The default is `30000`.
`AutoRedirect`	This 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`	Some servers require that client users can be authenticated. In the case of basic authentication, the server requires the client user to supply a username and password. This specifies the user name that is to be used for authentication. **Note:** Artix does not perform any validation on user names specified. It is the user's responsibility to ensure that user names are correct in terms of spelling and case (if case-sensitivity applies 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.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`Password`	Some servers require that client users can be authenticated. In the case of basic authentication, the server requires the client user to supply a username and password. This specifies the password that is to be used for authentication. **Note:** Artix does not perform any validation on passwords specified. It is the user's responsibility to ensure that passwords are correct in terms of spelling and case (if case-sensitivity applies 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.
`AuthorizationType`	Some servers require that client users can be authenticated. If basic username and password-based authentication is not in use by the server, this specifies the type of authentication that is in use. This specifies the name of the authorization scheme in use. 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, this does not need to be set. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Authorization`	Some servers require that client users can be authenticated. If basic username and password-based authentication is not in used by the server, this specifies the actual data that the server should use to authenticate the client. This 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. **Note:** If basic username and password-based authentication is being used, this does not need to be set. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`Accept`	This 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. 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 (that is, *) 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/`. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`AcceptLanguage`	This specifies what language (for example, American English) the client prefers for the purposes of receiving a response. Language tags are regulated by the International Organisation 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. 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`. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`AcceptEncoding`	This 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. Content codings are regulated by the Internet Assigned Numbers Authority (IANA). See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html` for more details of content codings. Possible content coding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`. 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.
`ContentType`	This is relevant if the client request specifies the `POST` method, to send data to the server for processing. This specifies the media type of the data being sent in the body of the client request. 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`. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Host`	This 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 this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`Connection`	This 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 `close`, to close the connection to the server after each request/response dialog. If `Keep-Alive` is specified, and the server honors it, the connection is reused for subsequent request/response dialogs. **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. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`ConnectionAttempts`	This specifies the number of times a client will transparently attempt to connect to server.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`CacheControl`	This 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. Valid values are: • `no-cache`—This prevents a cache from using 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`—This indicates that a cache must not store any part of a response or any part of the request that evoked it. • `max-age`—This indicates that the client can accept a response whose age is no greater than the specified time in seconds. • `max-stale`—This indicates that 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`—This indicates that the client wants a response that will be still be fresh for at least the specified number of seconds indicated by the value set for min-fresh. • `no-transform`—This indicates that a cache must not modify media type or location of the content in a response between a server and a client. • `only-if-cached`—This indicates that a cache should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
	• `cache-extension`—This indicates additional extensions to the other cache directives. Extensions might be informational (that is, do not require a change in cache behavior) or behavioral (that is, act as modifiers to the existing base of cache directives). 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. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Cookie`	This specifies the cookie to be sent to the server. Some session designs that maintain state use cookies to identify sessions. **Note:** If the cookie is static, you can supply it here. However, 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. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`BrowserType`	This 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. Specifying the browser type is usually only necessary if sites have HTML customized for use with Netscape as opposed to Internet Explorer, and so on. However, you can also specify the browser type to facilitate optimizing for different SOAP stacks. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`Referer`	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`	This 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.
`ProxyUserName`	This is only relevant if a proxy server exists along the message path. Some proxy servers require that client users can be authenticated regardless of whether those users have already been authenticated by any downstream login. In the case of basic authentication, the proxy server requires the client user to supply a username and password. This specifies the user name that is to be used for authentication. **Note:** Artix does not perform any validation on user names specified. It is the user's responsibility to ensure that user names are correct in terms of spelling and case (if case-sensitivity applies at application level).

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`ProxyPassword`	This is only relevant if a proxy server exists along the message path. Some proxy servers require that client users can be authenticated regardless of whether those users have already been authenticated by any downstream login. In the case of basic authentication, the proxy server requires the client user to supply a username and password. This specifies the password that is to be used for authentication. **Note:** Artix does not perform any validation on passwords specified. It is the user's responsibility to ensure that passwords are correct in terms of spelling and case (if case-sensitivity applies at application level).
`ProxyAuthorizationType`	This is only relevant if a proxy server exists along the message path. Some proxy servers require that client users can be authenticated regardless of whether those users have already been authenticated by any downstream login. If basic username and password-based authentication is not in use by the proxy server, this specifies the type of authentication that is in use. This specifies the name of the authorization scheme in use. 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`	This is only relevant if proxy servers are in use along the request-response chain. Some proxy servers require that client users can be authenticated regardless of whether those users have already been authenticated by any downstream login. If basic username and password-based authentication is not in used by the proxy server, this specifies the actual data that the proxy server should use to authenticate the client. This 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. **Note:** If basic username and password-based authentication is being used by the proxy server, this does not need to be set.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`UseSecureSockets`	This indicates whether the client wants to open a secure connection (that is, HTTP running over 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.
`ClientCertificate`	This is only relevant if the HTTP connection is to run securely over SSL or TLS (that is, if `UseSecureSockets` is set to `true`). This specifies the full path to the PEM-encoded X509 certificate issued by the certificate authority for the client. For example: `c:\aspen\x509\certs\key.cert.pem` Some servers might require the client to present a certificate, so that the server can authenticate the client.
`ClientCertificateChain`	This is only relevant if the HTTP connection is to run securely over SSL or TLS (that is, if `UseSecureSockets` is set to `true`). PEM-encoded X509 certificates can be issued by intermediate certificate authorities that are not trusted by the server, but which have had their certificates issued in turn by a trusted certificate authority. If this is the case, you can use `ClientCertificateChain` to allow the certificate chain of PEM-encoded X509 certificates to be presented to the server for verification. This specifies the full path to the file that contains all the certificates in the chain. For example: `c:\aspen\x509\certs\key.cert.pem`
`ClientPrivateKey`	This is only relevant if the HTTP connection is to run securely over SSL or TLS (that is, if `UseSecureSockets` is set to `true`). This is used in conjuction with `ClientCertificate`. It specifies the full path to the PEM-encoded private key that corresponds to the X509 certificate specified by `ClientCertificate`. For example: `c:\aspen\x509\certs\privkey.pem` This is required if, and only if, `ClientCertificate` has been specified.

Table 24: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`ClientPrivateKeyPassword`	This is only relevant if the HTTP connection is to run securely over SSL or TLS (that is, if `UseSecureSockets` is set to `true`). This specifies a password that is used to decrypt the PEM-encoded private key, if it has been encrypted with a password. The certificate authority typically encrypts these keys when sending them over a public network, and the password is delivered by a secure means. **Note:** Artix does not perform any validation on passwords specified. It is the user's responsibility to ensure that passwords are correct in terms of spelling and case (if case-sensitivity applies at application level).
`TrustedRootCertificate`	This is only relevant if the HTTP connection is to run securely over SSL or TLS (that is, if `UseSecureSockets` is set to `true`). This specifies the full path to the PEM-encoded X509 certificate for the certificate authority. For example: `c:\aspen\x509\ca\cacert.pem` This is used to validate the certificate presented by the server.

HTTP Transport Attributes

Overview

One of the basic properties of HTTP is that client or server information, and information about the possible content of a message, is made available through a series of header fields on an HTTP message. This section outlines both the client transport attributes and server transport attributes that can be sent, using Artix, in an HTTP request or response message.

In this section

This section discusses the following topics:

Transport Attributes Overview

Overview

This subsection outlines the background to the HTTP transport attributes that can be used with Artix.

What are transport attributes?

A number of the configuration attributes described in "HTTP WSDL Extensions" on page 187 can be subsequently transmitted, for information purposes, as transport attributes in the header of HTTP request and response messages. Client configuration attributes can be sent by the client as server transport attributes in the header of a request message. Similarly, server configuration attributes can be sent by the server as client transport attributes in the header of a response message.

Note: Transport attributes can only be sent if they have been configured as extensions to a WSDL contract, as described in "HTTP WSDL Extensions" on page 187.

Programmatic use of transport attributes

The application runtime can read transport attributes to facilitate it in the processing of client requests and server responses. See the *C++ Artix Programmer's Guide* for more details of how applications can handle transport attributes.

Server Transport Attributes

Overview

This subsection outlines the attributes that can be sent to a server for information purposes in the header of a request message.

Details

Table 25 describes the transport attributes that can be sent from a client to a server in the header of a request message.

Table 25: *HTTP Server Transport Attributes (Sheet 1 of 2)*

Configuration Attribute	Explanation
`UserName`	This lets the server know the user name of the browser user for the purposes of basic HTTP authentication by the server.
`Password`	This lets the server know the password of the browser user for the purposes of basic HTTP authentication by the server.
`AuthorizationType`	This lets the server know what type of authentication the client expects the server to use, if username and password-based basic authentication is not being used.
`Authorization`	This lets the server know the actual authentication data (authorization token) being sent by the client, if username and password-based basic authentication is not being used.
`Accept`	This lets the server know what multimedia (MIME) types (for example, text/html, image/gif, image/jpeg, and so on) the client can accept.
`AcceptLanguage`	This lets the server know what language(s) (for example, English, French, German, and so on) the client prefers for the purposes of receiving a request.
`AcceptEncoding`	This lets the server know what content codings (for example, gzip) the client can accept.
`ContentType`	If a client request is using the `POST` method, to send data to the server for processing, this lets the server know the MIME type of the data being sent. **Note:** This should be `text/xml` for web services. If the client is sending form data, this can be set to `application/x-www-form-urlencoded`.

Table 25: *HTTP Server Transport Attributes (Sheet 2 of 2)*

Configuration Attribute	Explanation
`Host`	This lets the server know what host the client prefers for clusters (that is, for virtual servers mapping to the same IP).
`Connection`	This lets the server know whether the client wants a particular connection to be kept open or not 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`	This lets the server know what behavior the client expects caches involved in the request chain to adhere to. See "CacheControl" on page 202 for more details of possible settings for this field.
`Cookie`	This lets the server know what cookie is being sent to the server. **Note:** This relates to static cookies. Dynamic cookies are set by the server when the server is first accessed, and are then handled automatically by the application runtime.
`BrowserType`	This lets the server know details about the browser from which the client request originates.
`Referer`	If the client request has resulted from the browser user clicking on a hyperlink rather than entering a URL from the keyboard, this lets the server know the URL that contains the hyperlink. This in turn lets the server generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. **Note:** This is sent automatically if the client request is configured (via the `AutoRedirect` attribute) to be automatically redirected when the server issues a redirection reply via the `RedirectURL` server-side attribute. This can allow the server to optimize processing based upon previous task flow. However, it is typically not used in web services applications.
`ClientCertificate`	If the HTTP connection is running securely over SSL or TLS, this lets the server know the PEM-encoded X509 certificate issued by the certificate authority for the client. Some servers can require the client to present a certificate, so that the server can authenticate the client.

Client Transport Attributes

Overview

This subsection outlines the attributes that can be sent to a client for information purposes in the header of a response message.

Details

Table 25 describes the transport attributes that can be sent from a server to a client in the header of a response message.

Table 26: *HTTP Client Transport Attributes*

Configuration Attribute	Explanation
`RedirectURL`	This lets the client know the URL to which the client request was redirected if the URL specified in the client request was 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 in the first line of the response is set to `302` and the status description is set to `Object Moved`.
`CacheControl`	This lets the client know what behavior the server expects caches involved in the response chain to adhere to. See "CacheControl" on page 192 for more details of possible settings for this field.
`ContentLocation`	This lets the client know the URL from which the requested resource is coming.
`ContentType`	This lets the client know the MIME type (that is, text/html, image/gif, image/jpeg, and so on) of the information that is being sent by the server.
`ContentEncoding`	This lets the client know how the information being sent by the server is encoded. This in turn lets the client know what decoding mechanisms it needs to retrieve the information.
`ServerType`	This lets the client know what type of server is sending the information.

CHAPTER 12

Working with IIOP Tunnels

IIOP tunnels provide access to CORBA services while using non-CORBA payload formats.

In this chapter

This chapter discusses the following topics:

Introduction to IIOP Tunnels

Overview

An IIOP tunnel provides a means for taking advantage of existing CORBA services while transmitting messages using a payload format other than CORBA. 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: IIOP tunneling is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports IIOP tunneling.

Benefits

Using IIOP tunnels provides the following benefits:

- Endpoints can publish their addresses in a CORBA naming service or a CORBA trader service
- Active connection management
- Transport level security
- Codeset negotiation
- Persistence

Supported payload formats

IIOP tunnels can transport messages using the following payload formats:

- SOAP
- Fixed format
- Fixed record length
- G2++
- Octet streams

Configuring the Artix to use IIOP tunnels

IIOP tunnels require that the OTS plug-in is loaded by Artix at start-up. To ensure that the OTS plug-in is loaded edit your application's orb plug-ins list to include `ots`. For more information on Artix configuration, see *Deploying and Managing Artix Solutions*.

Modifying a Contract to Use an IIOP Tunnel

Overview

Service Access Points (SAPs) that use IIOP tunnels require that a special port be added to the physical portion of the Artix contract.The port definition specifies the IOR used to locate the CORBA object and any POA policies the used in exposing the IIOP tunnel.

IIOP tunnel ports are described using the IONA-specific WSDL elements `<iiop:address>` and `<iiop:policy>` within the WSDL `<port>` element, to specify how the IIOP tunnel is configured.

Address specification

The IOR, or address, of the IIOP tunnel 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
  ```

- 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 the *Artix Administration Guide*.

- 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 tunnel 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 a number of POA polices the Artix service will use when creating the IIOP tunnel. These policies include:

- POA Name
- Persistence
- ID Assignment

Setting these policies lets you exploit some of the enterprise features of IONA's Application Server Platform 6.0, such as load balancing and fault tolerance, when deploying an Artix integration project using the IIOP tunnel. For information on using these advanced CORBA features, see the Application Server Platform 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 tunnel, 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 tunnel'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`*.

Example

For example, an IIOP tunnel port for the `personalInfoLookup` binding would look similar to Example 111:

Example 111: *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 tunnel to be located in a file called `objref.ior`, and creates a persistent POA with an object id of `personalInfo` to configure the IIOP tunnel.

CHAPTER 13

Sending Messages using SOAP

The SOAP plug-in lets you configure an Artix integration solution to use the SOAP payload format for communication between distributed applications. This chapter first provides an introductory overview of SOAP. It then provides a description of the WSDL extensions involved in extending an Artix contract for SOAP. It outlines the XML types supported by SOAP in Artix.

In this chapter

This chapter discusses the following topics:

Overview of SOAP

Overview

This section provides an introductory overview of the simple object access protocol (SOAP) in terms of its purpose, how it evolved, the elements of a SOAP message, and how it handles (encodes) application data types.

In this section

This section discusses the following topics:

Note: A complete introduction to SOAP is outside the scope of this guide. For more details see the W3C SOAP 1.1 specification at http://www.w3.org/TR/SOAP/. IONA's Artix product supports only version 1.1 of the W3C SOAP specification.

Background to SOAP

Overview

This subsection discusses the purpose of SOAP and how it evolved. It discusses the following topics:

What is SOAP?

SOAP is a lightweight, XML-based protocol that is used for client-server communications on the World Wide Web. The primary function of SOAP is to enable access to distributed services and to facilitate the exchange of structured and typed information between peers across the Web.

With the evolution of the Web, and the ever-increasing need to do business more quickly and more proactively across it, there arose a need to have a dynamic, flexible, extensible, but standards-based system of communication between applications across the Internet. SOAP evolved as a solution to this need, by combining existing standards such as extensible markup language (XML) and the hypertext transfer protocol (HTTP).

SOAP is termed a *messaging* protocol. It is a framework for transporting client request and server response messages in the form of XML documents over (usually) the HTTP transport.

XML

XML is a simple form of standard generalized markup language (SGML). The purpose of a markup language is to facilitate preparation of electronic documents, by allowing information to be added to the document text that indicates the logical components of the document or how they are to be formatted. SGML describes the relationship between a document's content and its structure.

XML uses user-defined tags to describe the actual data elements contained within a web page or file. (This is unlike the hypertext markup language (HTML), which can only use a limited set of predefined tags to describe how the contents of a web page or file are to be formatted.) XML tags are

unlimited, because they can be defined at the user's discretion, depending on the data elements that need to be defined. This is why XML is termed *extensible*. XML processors now exist for any common platform or language.

XML and Unicode

XML works on the assumption that all character data belongs to the universal character set (UCS). UCS is more commonly known as *unicode*. This is a mechanism for setting up binary codes for text or script characters that relate to the principal written languages of the world. Unicode therefore provides a standard means of interchanging, processing, and displaying written texts in diverse languages. See http://www.unicode.org for details.

Because unicode uses 16 bits to represent a particular character, it can represent more than 65,000 different international text characters. This makes Unicode much more powerful than other text representation formats, such as ASCII (American standard code for information interchange), which only uses 7 bits to represent a particular character and can only represent 128 characters. Unicode uses a conversion method called UTF (universal transformation format) that can convert text to 8–bit or 16–bit Unicode characters. To this effect, there are UTF–8 and UTF–16 encoding formats. All XML processors, regardless of the platform or programming language for which they are implemented, must accept character data encoded using UTF–8 or UTF–16 encoding formats.

HTTP

HTTP is the standard TCP/IP-based transport used for client-server communications on the Web. Its main function is to establish connections between distributed web browsers (clients) and web servers for exchanging files and possibly other information across the Internet. HTTP is available on all platforms, and HTTP requests are usually allowed through security firewalls. See "Working with HTTP" on page 179 for a more detailed overview of HTTP.

Given the dynamic features of XML and HTTP, SOAP has therefore become regarded as the optimum tool for enabling communication between distributed, heterogeneous applications over the Internet.

Note: Although most implementations of SOAP are HTTP-based, SOAP can be used with any transport that supports transmission of XML data. Depending on the particular transport in use, SOAP can also be implemented to support different types of message-exchange patterns, such as one-way or request-response.

SOAP specification

SOAP is a framework for transporting client request and server response messages in the form of XML documents over HTTP or some other transport. The W3C SOAP specification at http://www.w3.org/TR/SOAP/ defines the standards for SOAP in relation to:

- Format and components of SOAP messages.
- SOAP usage with HTTP.
- SOAP encoding rules for application-defined data types.
- SOAP standards for representing remote procedure calls (RPCs) and responses.

"SOAP Messages" on page 224 briefly discusses the format and components of SOAP messages, and their use with HTTP. "SOAP Encoding of Data Types" on page 230 briefly discusses how data types are handled in SOAP. Again, a complete introduction to these topics is outside the scope of this guide, and you should see the W3C SOAP 1.1 specification at http://www.w3.org/TR/SOAP/ for full details.

SOAP Messages

Overview

This subsection uses an example of a simple client-server application to outline the typical format of a SOAP request and response message. It discusses the following topics:

Example overview

The distributed application in this example involves a client that invokes a `GetStudentGrade` method on a target server. The client passes a student code and subject name, both of type `string`, as input parameters to the method request. On processing the request, the server returns the grade achieved by that student for that subject—the grade is of type `int`. The following example shows the logical definition of this application in a WSDL contract:

Example 112: *Example of logical definition in WSDL*

```
…
<message name="GetStudentGrade">
    <part name="StudentCode" type="xsd:string"/>
    <part name="Subject" type="xsd:string"/>
</message>
<message name="GetStudentGradeResponse">
    <part name="Grade" type="xsd:int"/>
</message>
<portType name="StudentPortType">
    <operation name="GetStudentGrade">
        <input message="tns:GetStudentGrade" name="GetStudentGrade"/>
        <output message="tns:GetStudentGradeResponse" name="GetStudentGradeResponse"/>
    </operation>
</portType>
…
```

Example of SOAP request message

Example 113 shows an example of the format of a typical SOAP request message, based on Example 112 on page 224 (in this case, the client has passed student code `815637` and subject `History` as input parameters):

Example 113: *Example of a SOAP Request Message*

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<?xml version="1.0" encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
      encoding/"/>
      <SOAP-ENV:Body>
          <m:GetStudentGrade xmlns:m="Some-URI">
              <StudentCode>815637</StudentCode>
              <Subject>History</Subject>
          </m:GetStudentGrade>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Explanation of SOAP request message

Example 113 on page 225 can be explained as follows:

1. The first five lines represent HTTP header information (in this example, the SOAP request is running over HTTP). When a SOAP request is running over HTTP, the HTTP method must be set to `POST`, the HTTP `Content-Type` header must be set to `text/xml`, and a `SOAPAction` HTTP header should also be included that specifies a URI indicating what is being requested. (However, the `SOAPAction` field can be left blank, in which case the URI specified in the first couple of lines is taken to indicate the intent of the request instead.)

   Note: See "Working with HTTP" on page 179 for more details of the format of HTTP request headers.

2. The SOAP Envelope is the top-level element and is mandatory in every SOAP message. It defines a framework for describing what is in the message and how to process it.
3. The SOAP Body element is mandatory in every SOAP message. It holds the actual message data in sub-elements called body entries. Each body entry relates to a particular data type and must be encoded as an independent element. Body entries can contain attributes called `encodingStyle`, `id`, and `href` (see "SOAP Encoding of Data Types" on page 230 for more details of these).

   In Example 113 on page 225, the SOAP Body contains two body entries, `StudentCode` and `Subject`, within a wrapper element that corresponds to the `GetStudentGrade` operation. The two body entries in this case correspond to the two input parameters for the `GetStudentGrade` operation.

Example of SOAP response message

Example 114 shows an example of the format of a typical SOAP response message, based on Example 112 on page 224 (in this case, the server has returned grade A):

Example 114:*Example of a SOAP Response Message*

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<?xml version="1.0" encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/"/>
        <SOAP-ENV:Body>
            <m:GetStudentGradeResponse xmlns:m="Some-URI">
                    <Grade>A</Grade>
            </m:GetStudentGradeResponse>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Explanation of SOAP response message

Example 114 can be explained as follows:

1. The first three lines represent HTTP header information (in this example, the SOAP response is running over HTTP). See "Working with HTTP" on page 179 for more details of the format of HTTP response headers.
2. The explanation of the SOAP Envelope element is the same as in "Explanation of SOAP request message" on page 225.
3. The explanation of the SOAP Body element is the same as in "Explanation of SOAP request message" on page 225, except in this case the SOAP Body contains one body entry, `Grade`, within a wrapper element that corresponds to the server response part of the `GetStudentGrade` operation. The body entry in this case corresponds to the output parameter returned by the server in response to the client request (that is, the grade for the student and subject combination specified by the client).

Example of SOAP response with fault

If an error occurs during the processing of a SOAP request, the server can handle and report the error within the SOAP Body of the response. Example 115 shows an example of the format of a typical SOAP response message indicating an error.

Example 115: *Example of SOAP Response with Error Information*

```
1  HTTP/1.1 500 Internal Server Error
   Content-Type: text/xml; charset="utf-8"
   Content-Length: nnnn

   <SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Body>
2        <SOAP-ENV:Fault>
             <faultcode>SOAP-ENV:Server</faultcode>
             <faultstring>Server Error</faultstring>
             <detail>
                 <e:myfaultdetails xmlns:e="Some-URI">
                     <message>
                         Application did not work
                     </message>
```

Example 115:*Example of SOAP Response with Error Information*

```
                    <errorcode>
                        1001
                    </errorcode>
                </e:myfaultdetails>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Explanation of SOAP response with fault

Example 115 on page 227 can be explained as follows:

1. The first three lines represent HTTP header information (in this example, the SOAP response is running over HTTP). See "Working with HTTP" on page 179 for more details of the format of HTTP response headers.
2. Errors are reported within a SOAP Fault element within the SOAP Body. In this case, the SOAP Body must not contain any other elements. Only one SOAP Fault element can be defined in any SOAP message. SOAP Fault in turn defines the following four sub-elements:

   `faultcode` This describes the error. The default faultcode values defined by the W3C SOAP specification are:

   - `VersionMismatch`—This means the SOAP Envelope was associated with an invalid namespace (that is, a namespace other than `http://schemas.xmlsoap.org/soap/envelope/`).
   - `MustUnderstand`—This means a header element that needed to be processed was not processed correctly.
   - `Client`—This means the message was not properly formed or did not contain appopriate information to be successfully processed.
   - `Server`—This means the message could not be processed, but not due to message contents.

   `faultstring` This provides a human-readable explanation of the fault.

`faultactor`	This indicates where the fault originated along the message path. This element is mandatory for an intermediary proxy application along the message path, but it is optional for the ultimate target server. **Note:** Artix supports the use of only one intermediary proxy along the message path. Example 115 on page 227 is an example of an error being reported by the ultimate target server, and it omits a `faultactor` attribute.
`detail`	This in turn contains sub-elements, called *detail elements*, that hold application-specific error information when the fault is due to unsuccessful processing of the SOAP Body.

SOAP Encoding of Data Types

Overview

This subsection provides an overview of the concepts of SOAP encoding. It discusses the following topics:

What is encoding?

Encoding is the process of converting application-defined data to binary form for transfer across a network. *Decoding* is the process of converting binary data back to an application-defined format. XML encoding and decoding rules, such as UTF-8 or UTF-16, define how data is to be converted between application-defined and binary form.

SOAP encoding rules define how application data types are to be structured in an XML document before being converted to binary. The overall process of encoding, data transfer, and subsequent decoding is termed *serialization*.

Role of SOAP encoding

XML uses the UTF-8 and UTF-16 encoding formats to convert data to binary form. As explained in "Background to SOAP" on page 221, all XML processors (regardless of platform or programming language) must accept character data encoded using UTF-8 or UTF1-16 formats.

Problems can arise, however, when converting data to and from binary, if the data is represented differently by different applications. For example, some systems might have an integer as a 32-bit value, while others might have it as a 16-bit value. Such disparities can lead to data corruption during the data conversion process.

To avoid potential data corruption due to differences between source and target systems, SOAP encoding and decoding rules are used as a stepping stone between the expression of data types in a particular programming language and the XML UTF-8 or UTF-16 encoding or decoding rules used to convert those data types to and from binary. (See Figure 4 on page 231 for

more details.) SOAP encoding rules, therefore, define the elements and data types that are designed to support serialization of data between disparate systems.

As shown in Figure 4, all data transferred as part of a SOAP payload is marshalled across the network as UTF-encoded binary strings.

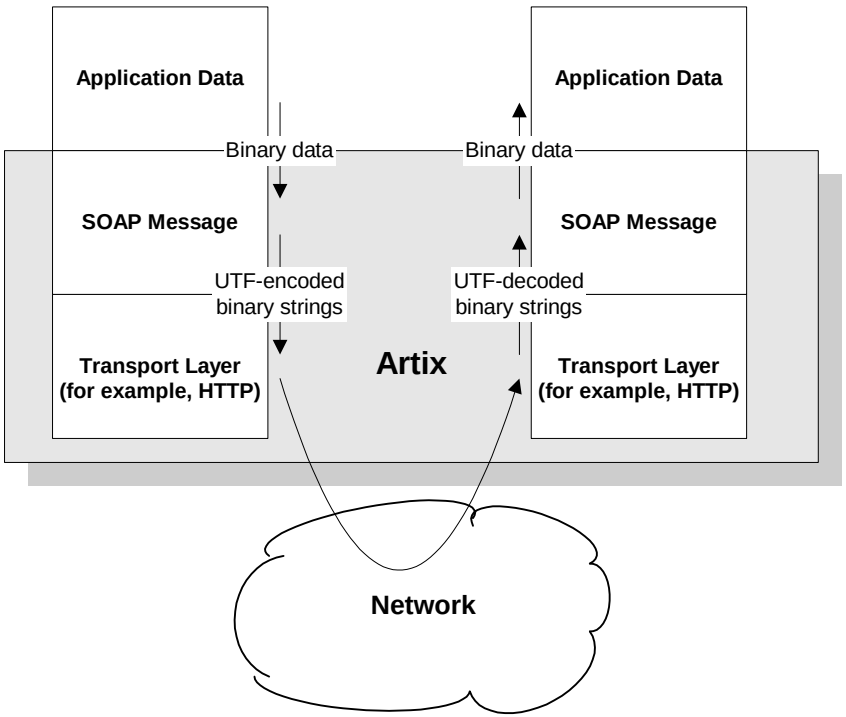

Figure 4: *Overview of Role of SOAP Encoding and Decoding*

SOAP encoding styles

A standard XML schema for SOAP encoding has been developed by the W3C and is located at http://schemas/xmlsoap/org/soap/encoding/. This W3C SOAP encoding schema uses the following namespace declaration:

```
xmlns:SOAP-ENC="http://schemas.xmlsoap/org/soap/encoding/"
```

It is recommended, but not mandatory, that a SOAP implementation adheres to the encoding style based on the W3C SOAP encoding schema. The W3C SOAP specification states that a company can use alternative encoding styles if it wants. To this effect, an `encodingStyle` attribute can be specified for any element within a SOAP message, to indicate the encoding rules that apply to that particular element.

An `encodingStyle` attribute can take one or more URIs as its value, with each URI denoting the location of a particular set of encoding rules. If specifying a list of URIs, each URI should be separated by a space. A list should also be ordered so that the URI relating to the most restrictive set of encoding rules is specified first, and the URI relating to the least restrictive set of encoding rules is specified last.

Encoding simple types

The W3C SOAP specification states that SOAP encodings can support all the simple types that are specified in the W3C *XML Schema Part 2: Datatypes* specification at `http://www.w3.org/TR/SOAP/#XMLS2`. In other words, a SOAP encoding should support any simple type that can be used in XML schema definition language.

The W3C SOAP encoding schema defines elements whose names correspond to each of the simple types defined in the W3C *XML Schema Part 2: Datatypes* specification. Among the simple types supported are integers, floats, doubles, booleans, and so on. Other types considered “simple” for the purposes of a SOAP encoding are strings, enumerations, and arrays of bytes.

In a SOAP encoding, each data value must be specified within an element. The type of a particular value can be denoted by the element name that encompasses it, provided that element name has been defined in the

encoding schema as a derived type. The following is an example of a schema fragment that defines a series of elements (for example, an element called `age` of type `int`, an element called `height` of type `float`, and so on):

```
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
   <simpleType base="xsd:string">
      <enumeration value="Blue"/>
      <enumeration value="Brown"/>
   </simpleType>
</element>
```

The following is an example of how the elements defined in the preceding sample schema might then be used in a SOAP encoding:

```
<age>34</age>
<height>6.0</height>
<displacement>-350</displacement>
<color>Brown</color>
```

If an element name in a SOAP encoding has not been defined as a derived type in an encoding schema (for example, the element name relating to a member of an array), that element must include an `xsi:type` attribute in the SOAP encoding to indicate the data type. See "Encoding complex array types" on page 236 for an example of this.

Encoding complex struct types

The W3C SOAP specification defines two complex data types—structs and arrays. A struct is a compound value whose members are each distinguished by a unique name (also known as that member's *accessor*).

The following is an example of a schema fragment that defines elements called `Book`, `Author`, and `Address` respectively, each of which is a structure containing a series of types:

```
<element name="Book">
<complexType>
    <sequence>
    <element name="title" type="xsd:string"/>
    <element name="author" type="tns:Author"/>
    </sequence>
</complexType>
</e:Book>
<element name="Author">
<complexType>
    <sequence>
    <element name="name" type="xsd:string"/>
    <element name="address" type="tns:Address"/>
    </sequence>
</complexType>
</e:Author>
<element name="Address">
<complexType>
    <sequence>
    <element name="street" type="xsd:string"/>
    <element name="city" type="xsd:string"/>
    <element name="country" type="xsd:string"/>
    </sequence>
</complexType>
</e:Address>
```

The following is an example of how the preceding schema definition could be subsequently used in a SOAP encoding (the following example shows embedded single-reference values for the author and address):

```
<e:Book>
    <title>Great Expectations</title>
    <author>
        <name>Charles Dickens</name>
        <address>
            <street>Whitechurch Road</street>
            <city>London</city>
            <country>England</country>
        </address>
     </author>
</e:Book>
```

In some cases an element might potentially contain more than one possible value. For example, if there was another book also called Great Expectations, written by some other author, there could be potentially more than one possible value for the author and address in the preceding example. When an element can contain more than one possible value it is termed *multireference*. In this case, an `id` attribute must be used to identify a multireference element, and a `href` attribute can be used to reference that element. For example, the `href` attribute of the `<author>` element in the following example refers to the `id` attribute of the multireference `<Person>` element. Similarly, the `href` attribute of the `<address>` element refers to the `id` attribute of the multireference `<Home>` element (this is assuming the author in question has more than one home).

```
<e:Book>
    <title>Great Expectations</title>
    <author href="#Person-1"/>
</e:Book>
<e:Person id="Person-1">
    <name>Charles Dickens</name>
    <address> href="Home-1"/>
</e:Person>
<e:Home id="Home-1"/>
    <street>Whitechurch Road</street>
    <city>London</city>
    <country>England</country>
</e:Home>
```

Encoding complex array types

The W3C SOAP specification defines two complex data types—structs and arrays. An array is a compound value whose member values are distinguished by means of ordinal position within the array. An array in SOAP is of type `SOAP-ENC:Array` or a type derived from that.

The following is an example (taken from the W3C SOAP specification) of a schema fragment that defines an element called `myFavoriteNumbers` that is of type `SOAP-ENC:Array`:

```
<element name="myFavoriteNumbers"
    type="SOAP-ENC:Array"/>
```

The following is an example (taken from the W3C SOAP specification) of how the array defined in the preceding sample schema could be subsequently used in a SOAP encoding:

```
<myFavoriteNumbers SOAP-ENC:arrayType="xsd:int[2]">
    <number>3</number>
    <number>4</number>
</myFavoriteNumbers>>
```

The preceding example shows an array of two integers, with both members of the array called `number` (this is unlike the members of a struct which must all have unique names). The members of a SOAP array do not have to be all of the same type. The following is an example of the SOAP encoding for an array where an `xsi:type` attribute is used to specify the type of each member of the array:

Note: As explained in "Encoding simple types" on page 232, if the type of a value is not identifiable from the element name (or accessor) corresponding to that value, an `xsi:type` attribute must be used in the SOAP encoding.

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:ur-type[4]">
  <thing xsi:type="xsd:int">98765</thing>
  <thing xsi:type="xsd:decimal">3.857</thing>
  <thing xsi:type="xsd:string">The cat sat on the mat</thing>
  <thing xsi:type="xsd:uriReference">http://www.iona.com</thing>
</SOAP-ENC:Array>
```

SOAP encoding rules also support:

- Arrays of complex structs or other arrays.
- Multi-dimensional arrays.
- Partially transmitted arrays.
- Sparse arrays.

See the W3C SOAP specification for more details of the encoding guidelines for arrays.

SOAP WSDL Extensions

Overview

This subsection provides an overview and description of the attributes that you can set as extensions to a WSDL contract for the purposes of using the SOAP payload format plug-in with Artix.

In this section

This section discusses the following topics:

Generating a SOAP Binding from a Logical Interface

Overview

Artix provides a command line tool, `wsdltosoap`, that will generate a SOAP binding from an existing logical interface defined in a WSDL `<portType>`. The tool will generate a new contract which includes the generated SOAP binding.

Using the tool

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}]
```

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`.

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

SOAP WSDL Extensions Overview

Overview

This subsection provides an overview of the WSDL extensions involved in configuring the SOAP payload format plug-in for use with Artix.

Configuration layout

Example 116 shows (in bold) the WSDL extensions used to configure the SOAP message format plug-in for use with Artix. (Ellipses (that is, …) are used to denotes sections of the WSDL that have been omitted for brevity.)

Example 116: *SOAP Configuration WSDL Extensions*

```
<definitions…
…
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap"
…

<definitions .... >
    <binding .... >
        <soap:binding style="rpc|document" transport="uri">
        <operation .... >
            <soap:operation soapAction="uri" style="rpc|document">
            <input>
                <soap:body use="literal|encoded" encodingStyle="uri-list">
            </input>
            <output>
                <soap:body use="literal|encoded" encodingStyle="uri-list">
            </output>
            <fault>*
                <soap:fault name="nmtoken" use="literal|encoded" encodingStyle="uri-list">
            </fault>
        </operation>
    </binding>

    <port .... >
        <soap:address location="uri"/>
    </port>
</definitions>
```

SOAP WSDL Extensions Details

Overview

This subsection describes each of the configuration attributes that can be set up as part of the WSDL extensions for configuring the SOAP message format plug-in for use with Artix. It discusses the following topics:

soap:binding element

The `soap:binding` element in a WSDL contract is defined within the `<binding>` component, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
```

Only one `soap:binding` element is defined in a WSDL contract. It is used to signify that SOAP is the message format being used for the binding. Table 27 describes the attributes defined within the `soap:binding` element.

Table 27: *Attributes for soap:binding*

Configuration Attribute	Explanation
`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.

Table 27: *Attributes for soap:binding*

Configuration Attribute	Explanation

	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 (based on the WSDL example in Example 112 on page 224) 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 (based on the WSDL example in Example 112 on page 224) is as follows if the style is document-based:
	`<SOAP-ENV:Body>`
	`  <StudentCode>815637</StudentCode>`
	`  <Subject>History</Subject>`
	`</SOAP-ENV:Envelope>`

`transport`	This 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

A soap:operation element in a WSDL contract is defined within an <operation> component, which is defined in turn within the <binding> component, 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. Table 27 describes the attributes defined within a soap:operation element.

Table 28: *Attributes for soap:operation*

Configuration Attribute	Explanation
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. See "soap:binding element" on page 241 for more details of the style attribute. The default value for soap:operation style is based on the value specified for the soap:binding 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

A soap:body element in a WSDL contract is defined within both the <input> and <output> components within an <operation> component, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
    <operation name="…">
        <soap:operation style="…" soapAction="…">
        <input>
            <soap:body use="…" encodingStyle="…" namespace="…">
        </input>
        <output>
            <soap:body use="…" encodingStyle="…" namespace="…">
        </output>
    </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 243, the structure of the SOAP Body within a SOAP message is dependent on the setting of the soap:operation style attribute.

Table 27 describes the attributes defined within the soap:body element.

Table 29: *Attributes for soap:body*

Configuration Attribute	Explanation
use	This 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. Valid values for soap:body use are encoded and literal.

Table 29: *Attributes for soap:body*

Configuration Attribute	Explanation
	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). The use attribute is mandatory.
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.

Table 29: *Attributes for soap:body*

Configuration Attribute	Explanation
`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.

soap:fault element

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 style` element in the WSDL is set to `document`, because faults do not contain parameters.

Table 27 describes the attributes defined within the soap:fault element.

Table 30: *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 244 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 245 for more details.

soap:address element

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>
```

Only one soap:address element is defined in a WSDL contract. It is only specified when you want to use SOAP over HTTP. If you want to use SOAP over a different transport (for example, IIOP), the element name in this case 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 the "Using the HTTP Plug-in" chapter of this guide for more details of http-conf:client and http-conf:server.

Table 27 describes the `location` attribute defined within the `soap:address` element.

Table 31: *Attribute for soap:address*

Configuration Attribute	Explanation

`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`
	The `soap:address` element is mandatory if you want to use SOAP over HTTP. It does not need to be set if you want to use SOAP over any other transport.

Supported XML Types

Overview

This section provides an overview of the XML data types that are supported by SOAP with Artix. It discusses the following topics:

- "Supported simple (built-in) types" on page 249.
- "Other supported types" on page 250.

Note: Artix does not currently support the use of multipart/related MIME attachments with SOAP.

Supported simple (built-in) types

The following simple (built-in) types are supported:

- `xsd:string`
- `xsd:int`
- `xsd:long`
- `xsd:short`
- `xsd:float`
- `xsd:double`
- `xsd:boolean`
- `xsd:byte`
- `xsd:decimal`
- `xsd:dateTime`
- `xsd:base64Binary`
- `xsd:hexBinary`

Other supported types

The following list provides an overview (and in some cases an example of) other supported types:

Type	Description/Example

Enumeration	For example:
	`<xsd:element name="EyeColor"`
	`type="EyeColorType"/>`
	`<xsd:simpleType name="EyeColorType" >`
	`<xsd:restriction base="xsd:string" >`
	`<xsd:enumeration value="Green" />`
	`<xsd:enumeration value="Blue" />`
	`<xsd:enumeration value="Brown" />`
	`</xsd:restriction>`
	`</xsd:simpleType>`

`<xsd:complexType>`	For example:
	`<xsd:complexType name="USAddress">`
	`<xsd:sequence>`
	`<xsd:element name="name" type="xsd:string"/>`
	`<xsd:element name="street" type="xsd:string"/>`
	`<xsd:element name="city" type="xsd:string"/>`
	`<xsd:element name="state" type="xsd:string"/>`
	`<xsd:element name="zip" type="xsd:decimal"/>`
	`</xsd:sequence>`
	`<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>`
	`</xsd:complexType>`
	Circular references that can occur with, for example, circular linked lists are not supported.

`xsd:attribute`	For example: `<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>`

Type	Description/Example

xsd:element	Occurence constraints (minOccurs and maxOccurs) for xsd:element within xsd:sequence. For example:
	`<xsd:complexType name="PurchaseOrderType">`
	`  <xsd:sequence>`
	`  <xsd:element name="shipTo"`
	`  type="USAddress"/>`
	`  <xsd:element name="billTo"`
	`  type="USAddress"/>`
	`  <xsd:element ref="comment"`
	`  minOccurs="0"/>`
	`  <xsd:element name="items"`
	`  type="Items"/>`
	`  </xsd:sequence>`
	`  <xsd:attribute name="orderDate"`
	`  type="xsd:date"/>`
	`</xsd:complexType>`

<xsd:ref>	Attribute for reference to global elements.

Derived simple types.	Derived simple types by restriction of an existing simple type. For example:
	`<xsd:simpleType name="myInteger">`
	`  <xsd:restriction base="xsd:integer">`
	`  <xsd:minInclusive value="10000"/>`
	`  <xsd:maxInclusive value="99999"/>`
	`  </xsd:restriction>`
	`</xsd:simpleType>`

Array derived from soap:Array.	Array derived from soap:Array by restriction using the wsdl:arrayType attribute. For example:
	`<complexType name="ArrayOfInteger">`
	`  <complexContent>`
	`  <restriction base="soapenc:Array">`
	`  <attribute`
	`  ref="soapenc:arrayType"`
	`  wsdl:arrayType="xsd:int[]"/>`
	`  </restriction>`
	`  </complexContent>`
	`</complexType>`

Type	Description/Example

`<xsd:sequence>`	For example:
	`<xsd:complexType name="PurchaseOrderType">`
	`  <xsd:sequence>`
	`  <xsd:element name="shipTo"`
	`  type="USAddress"/>`
	`  <xsd:element name="billTo"`
	`  type="USAddress"/>`
	`  <xsd:element name="items"`
	`  type="Items"/>`
	`  </xsd:squence>`
	`</xsd:complexType>`
	In this case, `minOccurs` and `maxOccurs` attributes are ignored.

`<xsd:choice>`	For example:
	`<xsd:complexType name="PurchaseOrderType">`
	`  <xsd:sequence>`
	`  <xsd:choice>`
	`  <xsd:group ref="shipAndBill"/>`
	`  <xsd:element name="singleUSAddress"`
	`  type="USAddress"/>`
	`  </xsd:choice>`
	`  <xsd:element name="items"`
	`  type="Items"/>`
	`  </xsd:sequence>`
	`</xsd:complexType>`
	In this case, `minOccurs` and `maxOccurs` attributes are ignored.

`<xsd:all>`	For example:
	`<xsd:complexType name="PurchaseOrderType">`
	`  <xsd:all>`
	`  <xsd:element name="shipTo"`
	`  type="USAddress"/>`
	`  <xsd:element name="billTo"`
	`  type="USAddress"/>`
	`  <xsd:element name="items"`
	`  type="Items"/>`
	`  </xsd:all>`
	`</xsd:complexType>`

Type	Description/Example

Complex type derived from simple type.	For example:
	`<xsd:element name="internationalPrice">`
	`<xsd:complexType>`
	`<xsd:simpleContent>`
	`<xsd:extension`
	`base="xsd:decimal">`
	`<xsd:attribute`
	`name="currency"`
	`type="xsd:string"/>`
	`</xsd:extension>`
	`</xsd:simpleContent>`
	`</xsd:complexType>`
	`</xsd:element>`

CHAPTER 14

Sending Messages as Fixed Record Length Data

Fixed record length data support allows Artix to interact with mainframe systems using COBOL.

Overview

Many applications send data in fixed length records. For example, COBOL applications often send fixed record data over WebSphere MQ. Artix provides a binding that maps logical messages to concrete fixed record length messages. The binding allows you to specify attributes such as encoding style, justification, and padding characters.

Type support

Artix supports text-based fixed length record data. For instance, numerals, such as 42, are represented as the ASCII characters `'4'` and `'2'`. This allows the data to be easily translated from one codeset to another if needed.

Binary data, such as packed decimals, are not supported.

In this chapter

This chapter discusses the following topics:

Fixed Record Length Message Data Mapping page 259

Creating a Fixed Binding from a COBOL Copybook

Overview

The primary use of the fixed binding is to work with systems built using COBOL. To facilitate the mapping of COBOL operations to Artix contracts, Artix provides a command line tool, `colboltowsdl`, that will import COBOL copybook data and generate an Artix contract containing a fixed binding to define the COBOL interface for Artix applications.

Using the tool

To generate an Artix contract from COBOL copybook data 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]
```

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 `<types>`. Default is to generate `<element>` for schema types.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`binding`*`.wsdl`.

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.

Fixed Record Length Message Data Mapping

Overview

Artix defines seven elements that extend the WSDL `<binding>` element to support the fixed record length binding. These elements are:

- <fixed:binding>
- <fixed:operation>
- <fixed:body>
- <fixed:field>
- <fixed:enumeration>
- <fixed:sequence>
- <fixed:choice>
- <fixed:case>

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 and add the following line to your contracts:

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

If you add a fixed record length binding to an Artix contract by hand you must also include this namespace.

<fixed:binding>

`<fixed:binding>` specifies that the binding is for fixed record length data. It has three optional attributes:

`justification`	Specifies the default justification of the data contained in the messages. Valid values are `left` and `right`. Default is `left`.
`encoding`	Specifies the codeset used to encode the text data. Valid values are any valid ISO locale or IANA codeset name. Default is `en`.
`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.

<fixed:body> takes three optional attributes:

justification	Specifies the default justification of the data contained in the messages. Valid values are left and right.
encoding	Specifies the codeset used to encode the 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.

These values override the defaults set in the <fixed:binding> element.

<fixed:body> will have one or more of the following child elements:

- <fixed:field>
- <fixed:sequence>
- <fixed:choice>

They describe the detailed mapping of the data to fixed length record data to be sent on the wire.

<fixed:field>

<fixed:field> is used to map simple data types to a fixed length record. Each <fixed:field> element has one required attribute, name, which corresponds to the name of the message part being mapped to the fixed record. This name must be the name of a message part defined in the logical message description.

Each <fixed:field> element that maps a message part also requires either the size attribute or the format attribute. A <fixed:field> element would never use both attributes.

size

size specifies the length of a string record. For example, the logical message part, raverID, described in Example 117 would be mapped to a <fixed:field> similar to Example 118.

Example 117:*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 118:*Fixed String Mapping*

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

format

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 119.

Example 119:*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 120. This provides Artix with the meta-data needed to properly handle the data.

Example 120:*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 121.

Example 121:*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 122.

Example 122:*Fixed Format Date Mapping*

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

bindingOnly

`<fixed:field>` elements supports an optional `bindingOnly` attribute. `bindingOnly` is a boolean attribute that specifies that the field is specific to the binding and does not appear in the logical message description. 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.

fixedValue

`fixedValue` can be used in place of the `size` and `format` attributes. It 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 121 on page 262, were mapped the the fixed field shown in Example 123, the actual message returned from the binding would always have the date 11/11/2112.

Example 123: *fixedValue Mapping*

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

<fixed:enumeration>

`<fixed:enumeration>` is a child element of `<fixed:field>` and is used to map enumerated types to a fixed record length message. It 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 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 124.

Example 124: *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 respresentations can be identical to the logical or some other value. The

enumerated type in Example 124 could be mapped to the fixed field shown in Example 125. Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to `OrangeTango`.

Example 125: *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:sequence>

`<fixed:sequence>` maps arrays and sequences to a fixed record length message. It has one required attribute, `name`, that corresponds to the name of the logical message part being mapped by this element.

`<fixed:sequence>` also takes two optional attributes, `occurs` and `counterName`. `occurs` specifies the number of times this sequence occurs in the message buffer. The default for `occurs` is `1`.

When you specify a value greater that `1` for `occurs`, you can also use `counterName`. `counterName` specifies the name of the field used for specifying the number of sequence elements are actually being sent in the message. The value of `counterName` corresponds to a `<fixed:field>` with at least enough digits to count to the value specified in `occurs` as shown in Example 126. 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 reciever so that the reciever will get the data in in the promised fixed format.

Example 126:*Using counterName*

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

A <fixed:sequence> can contain any number of <fixed:field>, <fixed:sequence>, or <fixed: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 <fixed:field> elements to fully describe the mapping of the data to the fixed record message. Example 127 shows an Artix contract fragment for such a mapping.

Example 127:*Mapping a Sequence to a Fixed Record Length Message*

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

Example 127:*Mapping a Sequence to a Fixed Record Length 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>
```

<fixed:choice>

<fixed:choice> is used to map unions into fixed record length messages. It takes one required attribute, name, which corresponds to the name of the logical message part being mapped.

<fixed:choice> also supports an optional attribute, discriminatorName, that specifies the message part used as the discriminator for the union. The value for discriminatorName corresponds to the name of a bindingOnly <fixed:field> that describes the type used for the union's descriminator as shown in Example 128. The only restriction in describing the descriminator 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 128 must be two digit integers.

Example 128:*Using discriminatorName*

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

A <fixed:choice> may contain one or more <fixed:case> child elements to map the cases for the union to a fixed record length message.

<fixed:case>

<fixed:case> is a child element of <fixed:choice> and describes the complete mapping of a union's individual cases to a fixed record length message. It takes two required attributes, name and fixedValue. name corresponds to the name of the case element in the union's logical description. fixedValue specifies the value of the descriminator that selects this case. The value of fixedValue must correspond to the format specified by the discriminatorName attribute of <fixed:choice>.

<fixed:case> must contain one child element to describe the mapping of the case's data to a fixed record length message. Valid child elements are <fixed:field>, <fixed:sequence>, and <fixed:choice>. Example 129 shows an Artix contract fragment mapping a union to a fixed record length message.

Example 129: *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>
```

Example 129:*Mapping a Union to a Fixed Record Length Message*

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

Example

Example 130 shows an example of an Artix contract containing a fixed record length message binding.

Example 130:*Fixed Record Length Message Binding*

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

Example 130:*Fixed Record Length Message Binding*

```
    <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 130:*Fixed Record Length Message Binding*

```
<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 130:*Fixed Record Length Message Binding*

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

CHAPTER 15

Sending Messages as Tagged Data

The Artix tagged data binding allows the use of self-describing messages.

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.

In this chapter

This chapter discusses the following topics:

Tagged Data Mapping

Overview

Artix defines seven elements that extend the WSDL binding element to support the tagged data format. These elements are:

- <tagged:binding>
- <tagged:operation>
- <tagged:body>
- <tagged:field>
- <tagged:enumeration>
- <tagged:sequence>
- <tagged:choice>
- <tagged:case>

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 and add the following line to your contracts:

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

If you add a tagged data binding to an Artix contract by hand you must also include this namespace.

<tagged:binding>

`<tagged:binding>` specifies that the binding is for tagged data format messages. It has ten attributes:

`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(%)`.
`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:

discriminator	Specifies a name to the operation 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. It has two attributes:

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

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 131.

Example 131: *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 132.

Example 132: *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. It has three attributes:

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 133 shows an Artix contract fragment for such a mapping.

Example 133:*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 133:*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. It takes three attributes:

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 unions 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 134 shows an Artix contract fragment mapping a union to a tagged data format.

Example 134:*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 134:*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 135 shows an example of an Artix contract containing a tagged data format binding.

Example 135:*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:fixed="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 135:*Tagged Data Format Binding*

```
      <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 135:*Tagged Data Format Binding*

```
<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 135:*Tagged Data Format Binding*

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

CHAPTER 16

Other Data Bindings for Sending Messages

Artix supports other data bindings such as G2++ and XML documents.

In this chapter

This chapter discusses the following topics:

G2++ Data 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 136:*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 137:*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 137:*WSDL Logical Description of ERecord Message*

```
<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 together to form the enclosing record (`PRequest`).

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

Example 138:*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> </G2Definitions>` tags. Each of the messages are defined with the scope of a `<G2MessageDescription> </G2MessageDescription>` construct. 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> </G2MessageComponent>` construct. 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>` scope (`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 139:*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 140:*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 141: *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` scope 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 142: *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.

Pure XML Format

Overview

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

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 `xmlformat` to represent the fixed record length extensions and add the following line to your contracts:

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

If you add an XML format binding to an Artix contract by hand you must also include this namespace.

Type support

The XML data format supports all of the types supported by the SOAP binding using doc/literal encoding. See "Supported XML Types" on page 249 for a full listing of the supported types.

Messages mapped to an XML format binding can only have one part. For example the message in Example 143 can be mapped to an XML format binding:

Example 143: *Valid XML Binding Message*

```
<message name="operator">
 <part name="lineNumber" type="xsd:int" />
</message>
```

However, the message in Example 144 cannot be mapped to an XML format binding because it has more than one part.

Example 144: *Invalid XML Binding Message*

```
<message name="matildas">
  <part name="dancing" type="xsd:boolean" />
  <part name="number" type="xsd:int" />
</message>
```

Mapping to an XML format binding

The XML format binding uses a single IONA-specific extension, `<xmlformat:binding>`, to identify the binding type. `<xmlformat:binding>` takes no attributes and is listed just after the `<binding>` element. Beyond the use of `<xmlformat:binding>`, an XML format binding is identical to a SOAP binding. Each operation is listed and its input, output, and fault messages are listed.

For example, Example 145 shows how the widget service would be mapped to an XML format binding.

Example 145: *XML Format Binding for Widgets*

```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
<binding name="widgetXMLBinding" type="tns:orderWidgets">
  <xmlformat:binding />
  <operation name="placeWidgetOrder">
    <input name="order" />
    <output name="bill" />
  </operation>
</binding>
```

Glossary

B

Binding

A binding associates a specific protocol and data format to operations defined in a portType.

C

Connection

An established communication link between any two Artix endpoints. Also the representation of such a link in System Designer, which displays connection characteristics such as its binding.

Contract

An Artix contract is a WSDL file that defines the interface and all connection (binding) information for that interface. 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’, ‘Type’, and ‘Schema.’

The physical contract defines the wire format, middleware transport, and service groupings, as well as the mapping between the portType ‘operations’ and wire formats, and the buffer layout for fixed formats and extensors, The physical contract defines: ‘Port,’ ‘Binding’ and ‘Service.’

D

Distillation

The process by which Artix helps the user reconcile type information among WSDL, message formats, and marshalling schemes. Artix supports only typed contracts, and type support for conversions is limited by the WSDL type meta-model and by the types supported for a specific marshalling. For example, ANYs are not supported in GIOP, and must be replaced with the typed data definition for the specific case.

E

Embedded Mode

Operational mode in which an application directly invokes Artix APIs. Code generated by System Designer is compiled into the application program. This provides the highest switch performance but is also the most invasive to the applications.

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).

H

Host

The network node on which a particular switch (service) resides. Also the representation of that node (in the context of an integration project) in Service Designer.

L

Language Binding

Support for a specified programming language, which allows Artix to generate server skeletons, client stubs, or both from a contract. Use of a language binding requires the Artix runtime to be linked with the application.

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 portType, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

R

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 an Artix standalone service. 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.

S

Service

An Artix service is an 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.

Standalone Mode

Operational mode in which an Artix switch runs in a separate process, and is invoked as a service. This is the least invasive approach but provides the lowest performance.

Switch

The implementation of an Artix WSDL service contract. Also the representation of such a service contract in System Designer.

System

A collection of services—for example, an WebSphere MQ system with several different queues on it.

T

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 a contract.

Index

U

V

W

X