IONA®

Artix™

User's Guide

Version 1.2, October 2003

Making Software Work Together™

Updated: 31-Oct-2003

M 3 1 1 3

Contents

List of Figures

List of Tables

Preface

Audience

This guide is intended for Artix System designers. It assumes that the reader has a working knowledge of the middleware transports that are being used to implement the Artix system. It also assumes that the reader is familiar with WSDL and software design concepts.

Organization of this guide

This guide is divided as follows:

- Chapter 1 provides an overview of the concepts behind using Artix to solve integration projects.
- Chapter 2 describes the use of Web Services Description Language and the specifics of Artix contracts.
- Chapter 3 describes how to configure Artix services to provide optimal performance.
- Chapter 4 describes how to deploy the Artix standalone service.
- Chapter 5 describes how to create message routes using Artix.
- Chapter 6 describes how to use the Artix Locator Service.
- Chapter 7 describes how to use the Artix Session Manager.
- Chapter 8 provides a detailed discussion of using the advanced logging features of Artix.
- Chapter 10 describes how to integrate CORBA systems into an Artix solution.
- Chapter 11 describes how to use HTTP with Artix.
- Chapter 12 describes how to integrate IBM WebSphere MQ systems into an Artix solution.

- Chapter 13 describes how to use FML and BEA Tuxedo in an Artix solution.
- Chapter 14 describes how to integrate TIBCO Rendezvous into an Artix solution.
- Chapter 16 describes how to use the different payload formats supported by Artix.

Online help

Artix includes comprehensive online help, providing:

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

The **Help** menu in Artix Designer provides access to this online help.

Related documentation

The document set for Artix includes the following:

- *Getting Started with Artix*
- *Artix Tutorial*
- *Artix User's Guide*
- *Artix C++ Programmer's Guide*
- *Artix Security Reference*
- *Artix Thread Library Reference*

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

Reading path

If you are new to Artix, you should read the documentation in the following order:

1. *Getting Started with Artix*

   The getting started book describes the basic concepts behind Artix. It also provides details on installing the system and a detailed walk through for developing a C++ client for a Web Service.

2. *Artix Tutorial*

   The tutorial guides you through programming Artix applications against all of the supported transports.

3. *The Artix User's Guide*

   The user's guide describes the development pattern for designing and deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.

4. *GUI Online Help*

   The Artix design tools have context sensitive online help that provides information specific to the tools that you are using.

5. *Artix C++ Programmer's Guide*

   The programmer's guide discusses the technical aspects of programming applications using the Artix C++ API.

Additional resources

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 guide 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 guide may use 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.

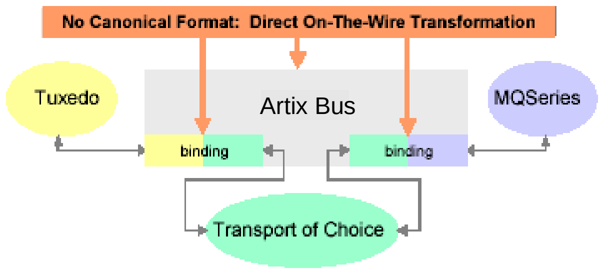


Figure 1: *Artix Message Transporting*

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 Artix Contracts” 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 5 on page 65.

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 Chapter 3 on page 27. 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*.

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:

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 inTable 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.

Artix Contract Specifics

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.

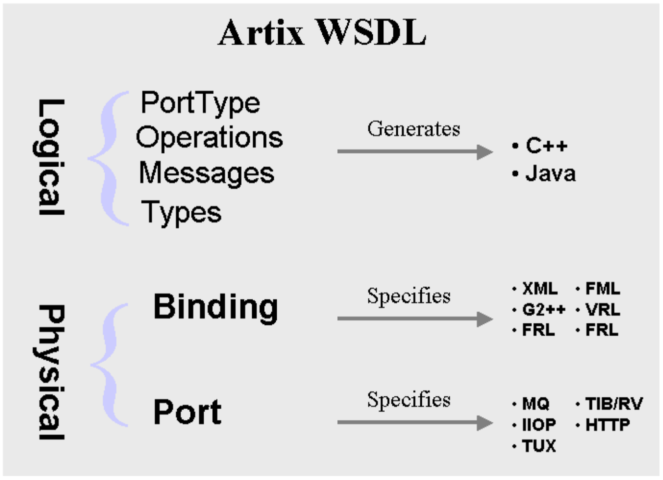


Figure 2: *An Artix Contract*

In this section

The following topics are discussed in this section:

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/routing`	Specifies the WSDL extensions to define routing between Artix SAPs.
`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 the *Artix C++ Programmer's Guide*.

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 172.

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 65.

CHAPTER 3

Configuration

Artix's customizable configuration provides a great deal of control over how Artix systems perform. Configuration settings affect the runtime behavior of Artix plug-ins.

Overview

There are several tasks involved in creating an environment in which Artix applications can run:

- Establishing the host computer environment
- Establishing the common and application-specific Artix runtime environments
- Configuring plug-ins to provide additional functions, for example, logging and routing plug-ins.

In this chapter

This chapter discusses the following topics:

Establishing the Host Computer Environment

Overview

To use the Artix design tools and the Artix runtime environment, the host computer must have several IONA specific environment variables set. These can be configured during installation or set later by running the provided `artix_env` script.

Environmental variables

Artix requires that the following environment variables be set on your system:

- JAVA_HOME
- IT_PRODUCT_DIR
- IT_CONFIG_FILE
- IT_IDL_CONFIG_FILE
- IT_CONFIG_DIR
- IT_CONFIG_DOMAINS_DIR
- IT_DOMAIN_NAME
- PATH

JAVA_HOME

The path to your system's JDK is specified with the system environment variable `JAVA_HOME`. This must be set if you wish to use the Artix Designer.

IT_PRODUCT_DIR

`IT_PRODUCT_DIR` points to the top level of your IONA product installaion. For example, if you install Artix into the `C:\Program Files\IONA` directory of your Windows system, you would set `IT_PRODUCT_DIR` to point to that directory.

Note: If you have other IONA products installed and you choose not to install them into the same directory tree, you will need to reset `IT_PRODUCT_DIR` each time you switch IONA products.

You can override this variable using the `-ORBproduct_dir` command line parameter when running your Artix applications.

IT_CONFIG_FILE

`IT_CONFIG_FILE` specifies the location of the configuration file Artix services use by default. You can overide this setting by using the `-ORBdomain_name` and `-ORBconfig_domains_dir` command line options.

IT_IDL_CONFIG_FILE

`IT_IDL_CONFIG_FILE` specifies the configuration used by the Artix IDL compiler. If this variable is not set, you will be unable to run the IDL to WSDL tools provided with Artix. The configuration file for the Artix IDL compiler is set as follows.

UNIX

Defaults to `$IT_INSTALL_DIR/artix/1.2/etc/idl.cfg`.

Windows

Defaults to `%IT_INSTALL_DIR%\artix\1.2\etc\idl.cfg`.

Note: Do not modify the default IDL configuration file.

IT_CONFIG_DIR

`IT_CONFIG_DIR` specifies the root configuration directory. The default root configuration directory is `/etc/opt/iona` on UNIX, and *`product-dir`*`\etc` on Windows. You can override this varaible using the `-ORBconfig_dir` command line parameter.

IT_CONFIG_DOMAINS_DIR

`IT_CONFIG_DOMAINS_DIR` specifies the directory where Artix searches for its configuration files. The configuration domains directory defaults to *`ORBconfig_dir`*`/domains` on UNIX, and *`ORBconfig_dir`*`\domains` on Windows. You can override this variable using the `-ORBconfig_domians_dir` command line parameter.

IT_DOMAIN_NAME

`IT_DOMAIN_NAME` specifies the name of the configuration domain used by Artix to locate its configuration information. This variable also specifes the file name the configuration information is stored in. For example the configuration information for domain `artix` would be stored in *`ORBconfig_dir`*`\domains\atrix.cfg` on Windows and *`ORBconfig_dir`*`/domains/artix.cfg` on Unix. You can override this variable with the `-ORBdomain_name` command line parameter.

PATH

The Artix bin directories should be placed first on the PATH, this ensures that the proper libraries, configuration files, and utility programs (for example, the IDL compiler) are used. These settings avoid problems that might otherwise occur if the Application Server Platform and/or Tuxedo (both of which include IDL compilers and CORBA class libraries) are installed on the same host computer.

The default Artix bin directories are:

UNIX

```
$IT_PRODUCT_DIR/artix/1.2/bin
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.2\bin
```

Running the artix_env Script

The installation process creates a script, `artix_env`, that captures the default information for setting the host computer's Artix environment. Running this script will properly configure your system to use Artix. It is located in the Artix bin directory.

```
IT_PRODUCT_DIR\artix\1.2\bin\artix_env
```

Configuring Artix Runtime Behavior

Overview

Artix, like the Application Server Platform, is built upon IONA's Adaptive Runtime Architecture (ART). Runtime behaviors are established through common and application-specific configuration settings that are applied during application startup. As a result, the same application code may be run -- and may exhibit different capabilities -- in different configuration environments.

An Artix configuration domain is a collection of configuration information in an Artix runtime environment. This information consists of configuration variables and their values. A default Artix configuration is provided when Artix is installed. The default configuration file is located in `%IT_PRODUCT_DIR%\artix\1.2\etc\domains\artix.cfg` on Windows and `$IT_PRODUCT_DIR/artix/1.2/etc/domains/artix.cfg` on Unix.

The contents of this file may need to be changed to modify Artix logging, routing, and other behaviors. Such changes may be the result of either automatically generated code, settings in the Artix System Designer, or manual editing of the Artix configuration file (`artix.cfg`).

You can also manually create new Artix configuration domains to compartmentalize your applications. However, this is only recommended if you are familiar with configuring IONA's ART platform.

Configuration Scopes

An Artix configuration is divided into scopes. These are typically organized into a hierarchy of scopes, whose fully-qualified names map directly to ORB names. By organizing configuration variables into various scopes, you can provide different settings for individual services, or common settings for groups of services.

Configuration scopes apply to a subset of services or to a specific services in an environment. Instances of the Artix standalone service can each have their own configuration scopes. A default Artix standalone service scope is automatically created when you install Artix.

Artix applications can have their own configuration scopes.

Application-specific configuration variables either override default values assigned to common configuration variables, or establish new configuration variables. Configuration scopes are localized through a name tag and delimited by a set of curly braces terminated with a semicolon, for example, `( nameTag {…}; )`.

A configuration scope may include nested configuration scopes. Configuration variables set within nested configuration scopes take precedence over values set in enclosing configuration scopes.

In the `artix.cfg` file, there are several predefined configuration scopes. For example, the `demo` configuration scope includes nested configuration scopes for some of the demo programs included with the product.

```
demo
{
 fml_plugin
 {
     orb_plugins = ["local_log_stream", "iiop_profile",
            "giop", "iiop", "soap", "http", "G2", "tunnel",
            "mq", "ws_orb", "fml"];
 };
 telco
 {
    orb_plugins = ["local_log_stream", "iiop_profile",
                   "giop”, "iiop”, "G2", "tunnel"];
    plugins:tunnel:iiop:port = "55002";
    poa:MyTunnel:direct_persistent = "true";
    poa:MyTunnel:well_known_address = "plugins:tunnel";

    server
      {
        orb_plugins = ["local_log_stream", "iiop_profile",
                     "giop", "iiop”, "ots", "soap", "http", "G2:,
                       "tunnel"];
        plugins:tunnel:poa_name = "MyTunnel";
     };
 };
 tibrv
 {
     orb_plugins = ["local_log_stream", "iiop_profile",
                    "giop", "iiop", "soap", "http", "tibrv"];

     event_log:filters = ["*=FATAL+ERROR"];
 };
};
```

Note that the `orb_plugins` list is redefined within each configuration scope.

Mapping to a configuration scope

To make an Artix process run under a configuration scope, you name that scope using the `-ORBname` parameter. During process initialization, Artix searches for a configuration scope with the same name as the `-ORBname` parameter.

There are two methods for supplying the `-ORBname` parameter to an Artix process:

- Pass the argument on the command line.
- Specify the ORBname as the third parameter to `IT_Bus::init()`.

For example, to start an Artix process using the configuration specified in the `demo.tibrv` configuration scope, you could start the process use the following syntax:

```
<processName> [application parameters] -ORBname demo.tibrv
```

Alternately, you could use the following code fragment to initialize the Artix bus:

```
IT_Bus::init (argc, argv, “demo.tibrv”);
```

If a corresponding configuration scope is not located, the process starts under the higher level configuration scope. If there are no configuration scopes that correspond to the `ORBname` parameter, the Artix process runs under the default global scope. For example, if the nested configuration scope `tibrv` does not exist, the Artix process uses the configuration specified in the `demo` configuration scope; if the scope `demo` does not exist, the process runs under the default global scope.

Namespaces

Most configuration variables are organized within namespaces, which serve to group related variables. Namespaces can be nested, and are delimited by colons (`:`). For example, configuration variables that control the behavior of a plug-in begin with `plugins:` followed by the name of the plug-in for which the variable is being set. For example, the to specify the port in which the Artix standalone service starts up on you would set the following variable:

```
plugins:artix_service:iiop:port
```

To set the location of the routing plug-in's contract you would set the following variable:

```
plugins:routing:wsdl_url
```

Variables

Configuration data is stored in variables that are set within each namespace. In some instances variables in different namespaces share the same variable names.

Variables can also be reset several times within successive layers of a configuration scope. Configuration variables set in narrower configuration scopes override variable settings in wider scopes. For example, a `company.operations.orb_plugins` variable would override a `company.orb_plugins` variable. Similarly, plug-ins specified at the `company` scope would apply to all SAPs in that scope, except those SAPs that belong specifically to the `company.operations` scope and its child scopes.

Data types

Each configuration variable has an associated data type that determines the variable's value. When creating configuration variables, you must specify the variable type.

Data types can be categorized into two types:

- Primitive types
- Constructed types

Primitive types

There are three primitive types: `boolean`, `double`, and `long`,.

Constructed types

Artix supports two constructed types: `string` and `ConfigList` (a sequence of strings).

- In an Artix configuration file, the `string` character set is ASCII.
- The `ConfigList` type is simply a sequence of `string` types. For example:

```
orb_plugins = ["local_log_stream", "iiop_profile",
    "giop","iiop"];
```

Runtime Configuration Variables

In this section

This section provides an overview of the most common configuration variables use by the Artix runtime. The following topics are discussed in this section:

ORB Plug-ins List

Overview

The orb_plugins variable specifies the plug-ins that Artix should load during application initialization. A plug-in is a class or code library that can be loaded into an Artix application at runtime. These plug-ins provide the user the ability to load network transports, payload format mappers, error logging streams, and other features "on the fly."

The default entry for the orb_plugins variable includes all of the logging and transport plug-ins:

```
orb_plugins = ["xmlfile_log_stream",
               "iiop_profile",
               "giop",
               "iiop",
               "soap",
               "http",
               "tunnel",
               "mq",
               "ws_orb"];
```

Artix plug-ins

Each network transport and payload format that Artix is capable of interoperating with uses its own plug-in. Many of the Artix features also use plug-ins. The Artix transport plug-ins are listed in Table 5.

Table 5: *Artix Transport Plug-ins*

Plug-in	Transport
http	Provides support for using HTTP and HTTPS.
ws_orb	Provides support for CORBA interoperability.
tunnel	Provides support for the IIOP transport using non-CORBA payloads.
tuxedo	Provides support for Tuxedo interoperability.
mq	Provides support for WebSphere MQ interoperability.
tibrv	Provides support for TIBCO Rendezvous interoperability.

The Artix payload format plug-ins are listed in Table 6.

Table 6: *Artix Payload Format Plug-ins*

Plug-in	Payload Format
`soap`	Decodes and encodes messages using the SOAP format.
`G2`	Decodes and encodes messages packaged using the G2++ format.
`fml`	Decodes and encodes messages packaged in FML format.
`fixed`	Decode and encodes fixed record length messages.

The Artix feature plug-ins are listed in Table 7.

Table 7: *Artix Service Plug-ins*

Plug-in	Artix Feature
`routing`	Enables Artix routing.
`locator_endpoint`	Enables endpoints to use the Artix locator service.
`locator_svr`	Enables the Artix locator. An Artix server acting as the locator service must load this plug-in.
`artix_wsdl_publish`	Enables Artix endpoints to publish and use Artix object references.

Binding Lists

Overview

The Artix binding namespace contains variables that specify interceptor settings. An interceptor acts on a message as it flows from sender to receiver. Computing concepts that fit the interceptor abstraction include transports, marshaling streams, transaction identifiers, encryption, session managers, message loggers, containers, and data transformers. Interceptors are a form of the “Chain of Responsibility” design pattern. Artix creates and manages chains of interceptors between senders and receivers, and the interceptor metaphor is a means of creating a “virtual connection” between a sender and a receiver.

The Artix binding namespace includes the following variables:

- client_binding_list
- server_binding_list

client_binding_list

Artix provides client request-level interceptors for OTS, GIOP, and POA collocation (where server and client are collocated in the same process), and message-level interceptors used in client-side bindings for IIOP, SHMIOP and GIOP.

The client_binding_list specifies a list of potential client-side bindings. Each item is a string that describes one potential interceptor binding. For example:

```
binding:client_binding_list = ["OTS+POA_Coloc","POA_Coloc","OTS+GIOP+IIOP","GIOP+IIOP"];
```

Interceptor names are separated by a plus (+) character. Interceptors to the right are “closer to the wire” than those on the left. The syntax is as follows:

- Request-level interceptors, such as GIOP, must precede message-level interceptors, such as IIOP.
- GIOP or POA_coloc must be included as the last request-level interceptor.
- Message-level interceptors must follow the GIOP interceptor, which requires at least one message-level interceptor.
- The last message-level interceptor must be a message-level transport interceptor, such as IIOP or SHMIOP.

When a client-side binding is needed, the potential binding strings in the list are tried in order, until one successfully establishes a binding. Any binding string specifying an interceptor that is not loaded, or not initialized through the `orb_plugins` variable, is rejected.

For example, if the `ots` plug-in is not configured, bindings that contain the `OTS` request-level interceptor are rejected, leaving `["POA_Coloc", "GIOP+IIOP", "GIOP+SHMIOP"]`. This specifies that POA collocations should be tried first; if that fails, (the server and client are not collocated), the `GIOP` request-level interceptor and the `IIOP` message-level interceptor should be used. If the `ots` plug-in is configured, bindings that contain the `OTS` request interceptor are preferred to those without it.

server_binding_list

`server_binding_list` specifies interceptors included in request-level binding on the server side. The POA request-level interceptor is implicitly included in the binding.

The syntax is similar to `client_binding_list`. However, in contrast to the `client_binding_list`, the left-most interceptors in the `server_binding_list` are “closer to the wire”, and no message-level interceptors can be included (for example, `IIOP`). For example:

```
binding:server_binding_list = ["OTS",""];
```

An empty string (`""`) is a valid server-side binding string; this specifies that no request-level interceptors are needed. A binding string is rejected if any named interceptor is not loaded and initialized.

The default `server_binding_list` is `["OTS", ""]`. If the `ots` plug-in is not configured, the first potential binding is rejected, and the second potential binding (`""`) is used, with no explicit interceptors added.

Thread Pool Control

Overview

Variables in the `thread_pool` namespace set policies related to thread control. They can be set globally for Artix instances in a configuration scope, or they can be set on a per-service basis. The settings set on a per-service basis override the global settings for the configuration scope.

To set the values globally, use the following syntax:

```
thread_pool:variable_name
```

To set the values on a per-service basis you can specify either the service's name or the service's fully qualified QName. The syntax is as follows:

```
thread_pool:service_name:variable_name
thread_pool:service_qname:variable_name
```

For example, if an Artix instance's contract has a service named `personalInfoService`, you would specify its thread control settings as follows:

```
thread_pool:personalInfoService:variable_name
```

The thread control settings specify the values for the thread pool on a per-port basis. For instance, if `personalInfoService` describes three ports, each port will have its own thread pool with values as specified by the settings in the `thread_pool:personalInfoService` namespace.

The following variables are in this namespace:

- `high_water_mark`
- `initial_threads`
- `low_water_mark`

high_water_mark

`high_water_mark` sets the maximum number of threads allowed in each port's thread pool. Defaults to `25`.

initial_threads

`initial_threads` sets the number of initial threads in each port's thread pool. Defaults to `2`.

low_water_mark

`low_water_mark` sets the minimum number of threads in each port's thread pool. Artix will terminate unused threads until only this number exists. Defaults to `5`.

Artix Plug-in Configuration

Overview

Each Artix transport and payload format have properties which are configurable. The variables used to configure plug-in behavior are specified in the configuration scopes of each Artix runtime instance and follow the same order of precedence. A plug-in setting specified in the global configuration scope will be overridden in favor of a value set in a narrower scope.

For example, if you set `plugins:routing:use_type_factory` to `true` in the global configuration scope and set it to `false` in the scope `widget_form`, all Artix runtimes, except for those running under the scope `widget_form`, would use `true` for the value of `use_type_factory`. Any Artix instance using the scope `widget_form` would use `false` the value of `use_type_factory`.

In this section

This section discusses the following topics:

Routing Plug-in

Overview

The routing plug-in uses the following variables:

- plugins:routing:shlib_name
- plugins:routing:routing_wsdl
- plugins:routing:use_type_factory
- plugins:routing:use_pass_through

plugins:routing:shlib_name

`plugins:routing:shlib_name` specifies the shared library that implements the routing plug-in. The default value for this is `it_routing`. Do not change this vaue.

plugins:routing:routing_wsdl

`plugins:routing:routing_wsdl` specifies the URL to search for Artix contracts containing the routing rules for your application. This value can be either a single URL or a list of URLs. If your application is using the routing plug-in you must spcecify a value for this variable.

plugins:routing:use_type_factory

`plugins:routing:use_type_factory` specifies if the routing plug-in loads user compiled type factories. The default setting is `false`.

Note: The use of type factories in routing is deprecated.

plugins:routing:use_pass_through

`plugins:routing:use_pass_through` specifies if the routing plug-in uses the pass-through routing optimization. This optimization allows the router to copy the message buffer directly from the source endpoint to the destination endpoint if both use the same binding. The default value is `true`.

Note: A few attributes are carried in the message body, as opposed to by the transport. Such attributes are always propagated when the pass-through optimization is in effect, regardless of attribute propagation rules.

CORBA Plug-in

Overview

In general, the Artix CORBA plug-in does not have any configuration variables directly associated with it. However, the CORBA plug-in is implemented using the same framework as IONA's Application Server Platform and it is affected by the same configuration settings as IONA's Application Server Platform.

For example, if you set the configuration variable:

```
policies:giop:interop_policy:send_principal = "true";
```

This will impact the CORBA messages that Artix sends.

Or, if you remove the plug-in `POA_Coloc` from the client binding list, then collocation will not work.

Shared library configuration

The only configuration variable that is directly associated with the CORBA plug-in is `plugins:ws_orb:shlib_name`. `plugins:ws_orb:shlib_name` specifies the shared library that implements the CORBA plug-in. The default value for this is `it_ws_orb`. Do not change this value.

TIBCO Rendezvous Plug-in

Overview

The TIBCO Rendezvous plug-in has only one configuration variable:

- plugins:tibrv:shlib_name

plugins:tibrv:shlib_name

`plugins:tibrv:shlib_name` specifies the shared library that implements the TIBCO Rendevous plug-in. The default value for this is `it_tibrv`. Do not change this vaue.

Tuxedo Plug-in

Overview

The Tuxedo plug-in has only one configuration variable:

- plugins:tuxedo:server

plugins:tuxedo:server

`plugins:tuxedo:server` is a boolean that specifies if the Artix process is a Tuxedo server and must be started using `tmboot`. The default is `false`.

Locator Service Plug-in

Overview

The locator service plug-in, `service_locator`, has the following configuration variables:

- plugins:locator:service_url
- plugins:locator:peer_timeout

plugins:locator:service_url

`plugins:locator:service_url` specifies the location of the Artix contract defining the location service and configuring its address. The name of this contract is `locator.wsdl`.

plugins:locator:peer_timeout

`plugins:locator:peer_timeout` specifies the amount of time, in milliseconds, the locator plug-in waits between keep-alive pings of the services registered with it. The default is 4000000 (4 sec.).

Locator Service Endpoint Plug-in

Overview

The locator service endpoint plug-in, `locator_endpoit`, has the following configuration variables:

- plugins:locator:wsdl_url
- plugins:session_endpoint_manager:peer_timout

plugins:locator:wsdl_url

`plugins:locator:wsdl_url` specifies the location of the Artix contract defining the location service and specifying the address locator endpoints use to communicate with the locator service. The name of this contract is `locator.wsdl`.

plugins:session_endpoint_manager:peer_timout

`plugins:session_endpoint_manager:peer_timout` specifies the specifies the amount of time, in milliseconds, the server waits between keep-alive oings of the locator service. The default is 4000000 (4 sec.).

Session Manager Plug-in

Overview

The session manager plug-in, `session_manager_service`, has the following configuration variables:

- plugins:session_manager_service:service_url
- plugins:session_manager_service:peer_timeout

plugins:session_manager_service:service_url

`plugins:session_manager_service:service_url` specifies the location of the Artix contract defining the session manager. The name of this contract is `session-manager.wsdl` and it is located in the wsdl folder of your installation.

plugins:session_manager_service:peer_timeout

`plugins:session_manager_service:peer_timeout` specifies the amount of time, in milliseconds, the session manager plug-in waits between keep-alive pings of the services registered with it. The default is 4000000 (4 sec.).

Session Manager Simple Policy Plug-in

Overview

The session manager's simple policy plug-in, `sm_simple_policy`, has the following configuration variables:

- plugins:sm_simple_policy:max_concurrent_sessions
- plugins:sm_simple_policy:min_session_timeout
- plugins:sm_simple_policy:max_session_timeout

plugins:sm_simple_policy:max_concurrent_sessions

`plugins:sm_simple_policy:max_concurrent_sessions` specifies the maximum number of concurrent sessions the session manager will allocate. Default value is `1`.

plugins:sm_simple_policy:min_session_timeout

`plugins:sm_simple_policy:min_session_timeout` specifies the minimum amount of time, in seconds, allowed for a session's timeout setting. Zero means the unlimited. Default is 5.

plugins:sm_simple_policy:max_session_timeout

`plugins:sm_simple_policy:max_session_timeout` specifies the maximum amount of time, in seconds, allowed for a session's timesout setting. Zero means the unlimited. Default is 600.

Session Manager Endpoint Plug-in

Overview

The session manager endpoint plug-in, `session_endpoint_manager`, has the following configuration variables:

- plugins:session_endpoint_manager:wsdl_url
- plugins:session_endpoint_manager:endpoint_manager_url
- plugins:session_endpoint_manager:default_group
- plugins:session_endpoint_manager:header_validation

plugins:session_endpoint_manager:wsdl_url

`plugins:session_endpoint_manager:wsdl_url` specifies location of the contract defining the session management service the endpoint manager is to contact.

plugins:session_endpoint_manager:endpoint_manager_url

`plugins:session_endpoint_manager:endpoint_manager_url` specifies the location of the contract defining the enpoint manager. The contract contains the contact information for the endpoint manager.

plugins:session_endpoint_manager:default_group

`plugins:session_endpoint_manager:default_group` specifies the default group name for all endpoints that are instantiated using the configuration scope.

plugins:session_endpoint_manager:header_validation

`plugins:session_endpoint_manager:header_validation` specifies whether or not a server validates the session headers passed to it by clients. Default value is `true`.

Artix Standalone Service

Artix lets you deploy middleware translation functions as a standalone service external to both client and server applications. The Artix standalone service can perform transport switching, message routing, and middleware bridging between non-Artix enabled applications.

In this chapter

This chapter discusses the following topics:

The Artix Standalone Service

Overview

The Artix standalone service is a minimally invasive means of connecting applications that use different communication transports and message formats. It does not require that any Artix-specific code be compiled or linked into existing applications.

How it works

The Artix standalone service is a daemon that listens for traffic on access points specified in the Artix contract. It re-directs messages based on the routing rules you provide, and performs any transport switching and message formatting needed for the receiving application. Neither application is aware that its messages are being intercepted by Artix and no application development is required.

Note: Artix requires that services being integrated use equivalent message layouts. For example, a service expecting a `long` cannot be sent a `float`.

The standalone service's behavior is controlled by a combination of an Artix contract and the Artix configuration file. For more information on Artix contracts see "Understanding Artix Contracts" on page 7. For more information on configuring the Artix runtime see "Configuration" on page 27.

Deployment patterns

An Artix standalone service can be deployed in a number of ways. Two common deployment patterns are:

Deploying several daemons, each of which bridges between two distinct applications.

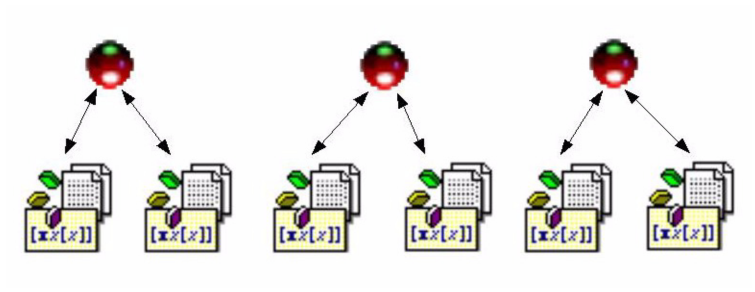


Figure 3: *Using multiple Artix daemons*

This approach simplifies designing integration solutions and provides faster processing of each message. Using this approach, the Artix contract describing the interaction of the applications is simpler because it contains only the logical interfaces shared by the two applications, and the bindings for each payload format.

Because most applications use only one network transport, the number of ports will be minimal and the routing rules will also be simple. The fact that the contract is kept simple also enhances the performance of each individual daemon because it has less processing to do. In this approach, each daemon's resource usage can also be limited by tailoring its configuration to optimize the daemon for the particular integration task for which it is responsible.

Deploying one daemon to bridge between all of the applications in a particular domain.

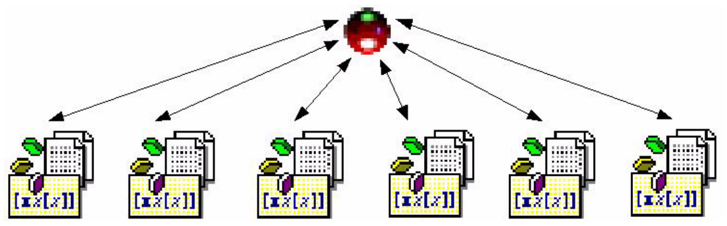


Figure 4: *Using a single Artix daemon*

This approach limits the number of external services required in your deployment environment. This can simplify monitoring and installation of deployments. It also reduces the number of “moving parts” in an integration solution.

Configuring the Service

Overview

Each instance of the Artix standalone service running on a host machine needs its own configuration scope to specify the unique port on which its administrative interface listens. Each instance also needs a corresponding administrative interface configuration scope.

Having separate configuration scopes for each instance of the service also allows greater control over the resources the service uses. You can specify that it only load the transport and payload format plug-ins it requires. You can also control the services threading and time-out behaviors.

For more information on configuring Artix, see "Configuration" on page 27.

Orb plugins list

In addition to the Artix plugins that provide support for the transports and payload formats it will be working with, the Artix standalone service needs to load the following plugins:

- iiop_profile
- logging
- iiop
- giop

These need to be entered in its `orb_plugins` list.

Service plug-in settings

The configuration variable that controls the behavior of the Artix standalone service are in the `plugins:artix_service` namespace. Table 8 lists the variables and their settings.

Table 8: *Artix Standalone Service Configuration Variables*

Variable	Effect
`shlib_name`	Specifies the name of the Artix service's shared library. This value should always be set to `it_artix_service_svr`.

Table 8: *Artix Standalone Service Configuration Variables*

Variable	Effect
`iiop:port`	Specifies the port number on which the service listens for calls from its administrative interface. See "Service admin interface".
`iiop:host`	Specifies the name of the host computer on which the service is running. See "Service admin interface".
`direct_persistence`	Specifies if the service's object reference is persistent across multiple invocations.

Service admin interface

Each instance of the Artix standalone service must have a corresponding administrative interface configuration scope. This scope must contain an entry for `initial_references:IT_ArtixServiceAdmin:reference`. `initial_references:IT_ArtixServiceAdmin:reference` specifies the port number of this admin interface's corresponding Artix service. The port number is specified using the `corbaloc` syntax:

```
corbaloc:iiop:1.2@hostname:port/IT_ArtixServiceAdmin
```

`hostname` is the hostname of the computer on which the corresponding Artix service is running. *`port`* is the port number on which the corresponding Artix service is listening.

Starting and Stopping the Service

Starting the service

To start the Artix standalone service, use the following script:

```
start_artix_service
```

This script starts an instance of the Artix standalone service using the default configuration scope of `iona_services.artix_service`.

Alternatively, you can start the service directly using the following command:

```
itartix_service -ORBname orb_name -ORBdomain_name domain_name
   -ORBconfig_domains_dir domain_dir run [-background]
```

Table 9 describes the parameters taken by `it_artix_service`.

Table 9: *it_artix_service Parameters*

Parameter	Description
`-ORBname` *`orb_name`*	Specifies the scope under which the service finds its configuration details.
`-ORBdomain_name` *`domain_name`*	Specifies the service's configuration file name. The configuration file has the name *`domain_name`*`.cfg`. For example, given domain name `acmewidgets`, the service will read its configuration from `acmewidgets.cfg`.
`-ORBconfig_domains_dir` *`domain_dir`*	Specifies the location of the service’s configuration file.
`run`	Specifies that the service is to begin monitoring.
`-background`	Specifies that the service is to run in the background. If this parameter is not specified, the service runs in the foreground of the active command window.

For more information about configuring Artix see “Configuration” on page 27.

Stopping the service

To stop the Artix standalone service use the following script:

```
stop_artix_service
```

This script will stop an instance of the Artix standalone service started using the start script, `start_artix_service`.

Alternatively, you can manually call the service's administrative interface to stop the service. To do so use the following command:

```
itartix_service_admin -ORBname orb_name
```

The value passed with the `-ORBname` flag specifies the configuration scope under which the administrative interface finds its configuration information. The vital entry in the administrative interfaces configuration is the entry for `initial_references:IT_ArtixServiceAdmin:reference`. This entry must contain the corbaloc address of the Artix service instance you wish to shutdown.

Installing the Service as a Windows Service

Overview

On Windows systems, you can install instances of the Artix standalone service as a service. This means the service starts at system boot and that limited management functionality is provided through the Windows service controls.

Installing the service

To install the Artix standalone service as a Windows service, use the following script:

```
install_artix_service
```

This script installs the Artix standalone service using the default configuration scope of `iona_services.artix_service`.

Alternatively, you can install an instance of the service directly using the following command:

```
it_artix_service -ORBname orb_name -ORBdomain_name domain_name
    -ORBconfig_domains_dir domain_dir install
```

Table 10 describes the parameters taken by `it_artix_service`.

Table 10: *it_artix_service Parameters*

Parameter	Description
`-ORBname` *`orb_name`*	Specifies the scope under which the service finds its configuration details.
`-ORBdomain_name` *`domain_name`*	Specifies the service's configuration file name. The configuration file has the name *`domain_name`*`.cfg`. For example, given domain name `acmewidgets`, the service will read its configuration from `acmewidgets.cfg`.
`-ORBconfig_domains_dir` *`domain_dir`*	Specifies the location of the service’s configuration file.
`install`	Specifies that the service is to installed as a Windows service.

Uninstalling the service

To uninstall the Artix standalone service as a Windows service use the following script:

```
uninstall_artix_service
```

This script uninstalls the Artix standalone service using the default configuration scope of `iona_services.artix_service`.

Alternatively, you can uninstall instances of the service directly using the following command:

```
it_artix_service -ORBname orb_name -ORBdomain_name domain_name
   -ORBconfig_domains_dir domain_dir uninstall
```

Table 10 describes the parameters taken by `it_artix_service`.

Table 11: *it_artix_service Parameters*

Parameter	Description
`-ORBname` *`orb_name`*	Specifies the scope under which the service finds its configuration details.
`-ORBdomain_name` *`domain_name`*	Specifies the service's configuration file name. The configuration file has the name *`domain_name`*`.cfg`. For example, given domain name `acmewidgets`, the service will read its configuration from `acmewidgets.cfg`.
`-ORBconfig_domains_dir` *`domain_dir`*	Specifies the location of the service’s configuration file.
`uninstall`	Specifies that the service is to remove itself from the Windows registry.

Contracts for the Standalone Service

Routing

Contracts for instances of the Artix standalone service must have routing rules to direct the flow of messages between the services defined within the contract.

You must also ensure that the routing plug-in is loaded by the Artix standalone service by placing the following entry in the `orb_plugins` list of the instance's configuration scope:

```
orb_plugins = [... "routing"];
```

Locating the contracts

The Artix standalone service loads the contract specified by the `plugins:routing:wsdl_url` configuration variable. For example if an instance of the Artix standalone service was designed to use a contract called `personalInfo.wsdl` and the contract was located in `/etc/contracts`, you would place the following in the instance's configuration scope:

```
plugins:routing:wsdl_url="/etc/contracts/personalInfo.wsdl";
```

For more information

For more information on Artix runtime configuration, see "Configuring Artix Runtime Behavior" on page 31.

CHAPTER 5

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.

Configuring Artix to Use Routing

Overview

Artix port- and operation-based routing is implemented as a plug-in to the Artix runtime. Content based routing does not require that the routing plug-in be loaded.

Adding the routing plug-in

When using Artix port- or operation-based routing you must add the routing plug-in to your SAP's `orb_plugin` list. The routing plug-in is simply called `routing`. The following shows an `orb_plugin` list for an Artix SAP that uses routing:

```
orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
               "iiop", "soap", "mq", "routing"];
```

See "Configuration" on page 27 for more information.

Locating the routing information

You need to add configuration information to point the routing plug-in to the contract, or contracts, that contain the routing information the router is to use. This is done with the `plugins:routing:wsdl` variable. This variable specifies the contracts the routing plug-in will parse for routing rules. The contract names are relative to the location from which the Artix SAP is started.

For example, if an SAP's configuration contained the following entry:

```
plugins:routing:wsdl=["route1.wsdl", "../route2.wsdl",
                      "/artix/routes/route3"];
```

The routing plug-in would expect that `route1.wsdl` was located in the directory in which the SAP was started and `route2.wsdl` was located one directory level higher.

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 69, 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 78.

<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 68.

`<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 68.

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 78.

`<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 69, 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 70.

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 70, to `getScore` defined in `finalScorePortType`, shown in Example 11 on page 69, your Artix contract would contain the elements in Example 14.

Example 14: *Operation to Operation Routing*

```
1 <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>
2 <routing:route name="scoreRoute">
    <routing:source service="tns:fullScoreService"
                    port="tns:fullScorePort"/>
    <routing:operation name="getFinalScore" target="getScore"/>
    <routing:destination service="tns:finalScoreService"
                         port="tns:finalScorePort"/>
  </routing:route>
```

There are two sections to the contract fragment shown in Example 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 69.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 68.

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 68.

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 "Using the HTTP Plug-in" on page 225. For information on the transport attributes for WebSphere MQ see "Using the WebSphere MQ Plug-in" on page 269.

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>
```

Routing with Artix Designer

Overview

The Artix Designer includes a routing wizard that assists you in creating routes from the services available in your contract. It walks you through the steps of creating a route and provides you with the valid options for the services available. It performs all of the compatibility testing for you and will never allow you to create an invalid route.

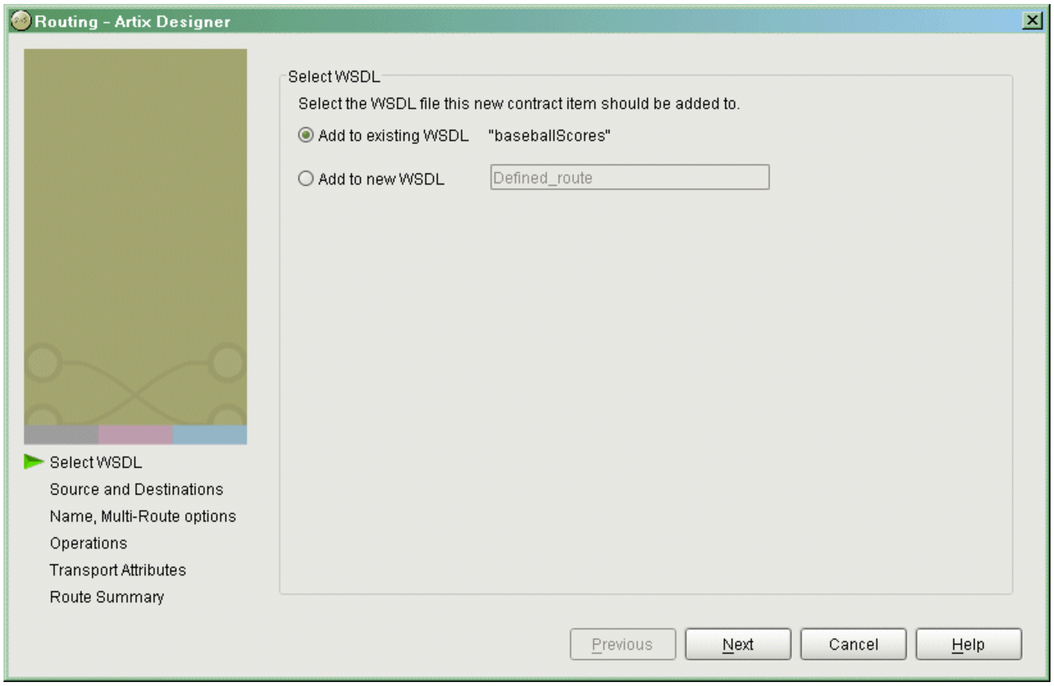
Creating a route

To create a route with the Artix Designer complete the following steps:

1. Load a contract with multiple service definitions that have operations that can be routed.
2. Select **Contracts|New|Route** from the Designer menu.

Note: If the Route option is not available, your contract does not have any compatible operations for routing.

3. You will see a screen like Figure 5.



Select WSDL

Select the WSDL file this new contract item should be added to.

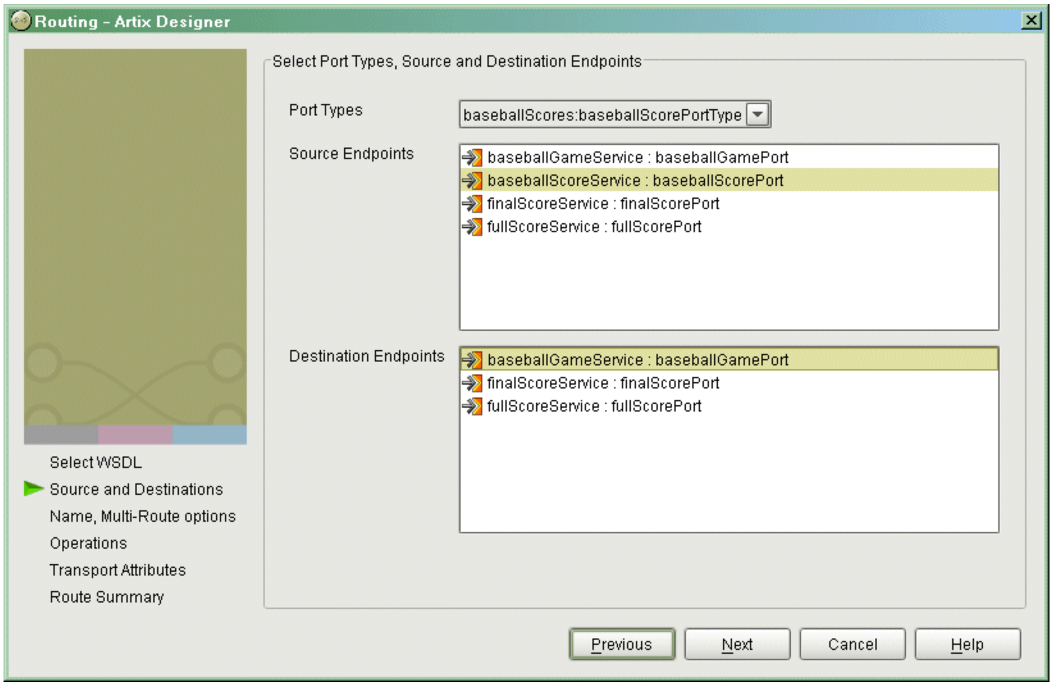
(●) Add to existing WSDL "baseballScores"

( ) Add to new WSDL Defined_route

Figure 5: *Routing WSDL Location*

4. Select where you want to add the routing information.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.

6. You will see a screen like Figure 6.



Select Port Types, Source and Destination Endpoints

Port Types: baseballScores:baseballScorePortType

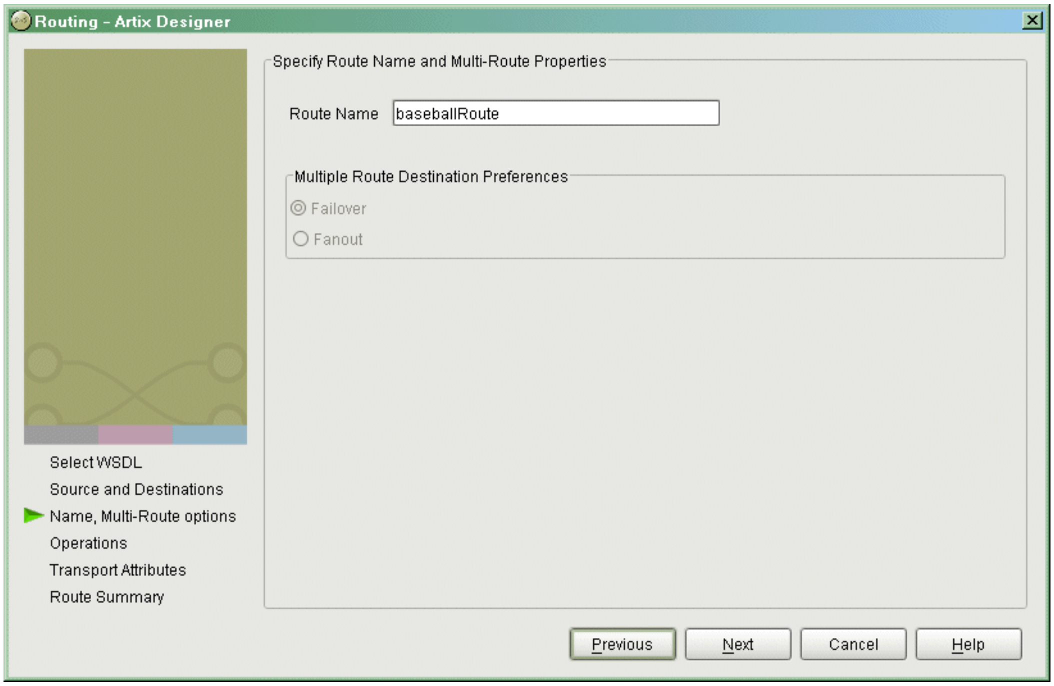
Source Endpoints:
- baseballGameService : baseballGamePort
- baseballScoreService : baseballScorePort
- finalScoreService : finalScorePort
- fullScoreService : fullScorePort

Destination Endpoints:
- baseballGameService : baseballGamePort
- finalScoreService : finalScorePort
- fullScoreService : fullScorePort

Figure 6: *Source and Destination Selection*

7. Select the source `portType` for the route from the **PortType** pull-down list.
8. Select the source endpoint from the available options in the **Source Endpoints** list.
9. Select the destination endpoint from the available options in the **Destination Endpoints** list.
10. Click **Next**.

11. You will see a screen like Figure 7.



Specify Route Name and Multi-Route Properties

Route Name: baseballRoute

Multiple Route Destination Preferences

[✓ Failover]

Fanout

Figure 7: *Route Properties*

12. Enter the name of your route in the Route Name field.
13. If you selected multiple destination endpoints on the previous screen, select either **Failover** or **Fanout** under **Multiple Route Destination Preference**.

    Note: This panel will allow you to select an invalid multiroute behavior and you will get an error dialog when you click **Next**.

14. Click **Next**.

15. You will see a screen like Figure 8.

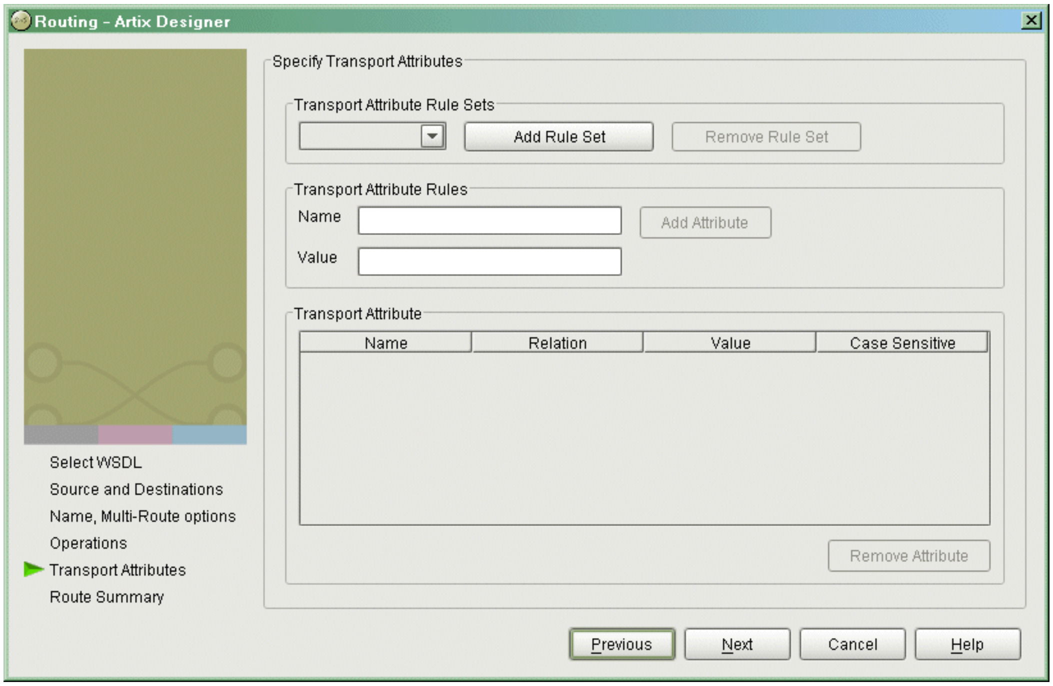


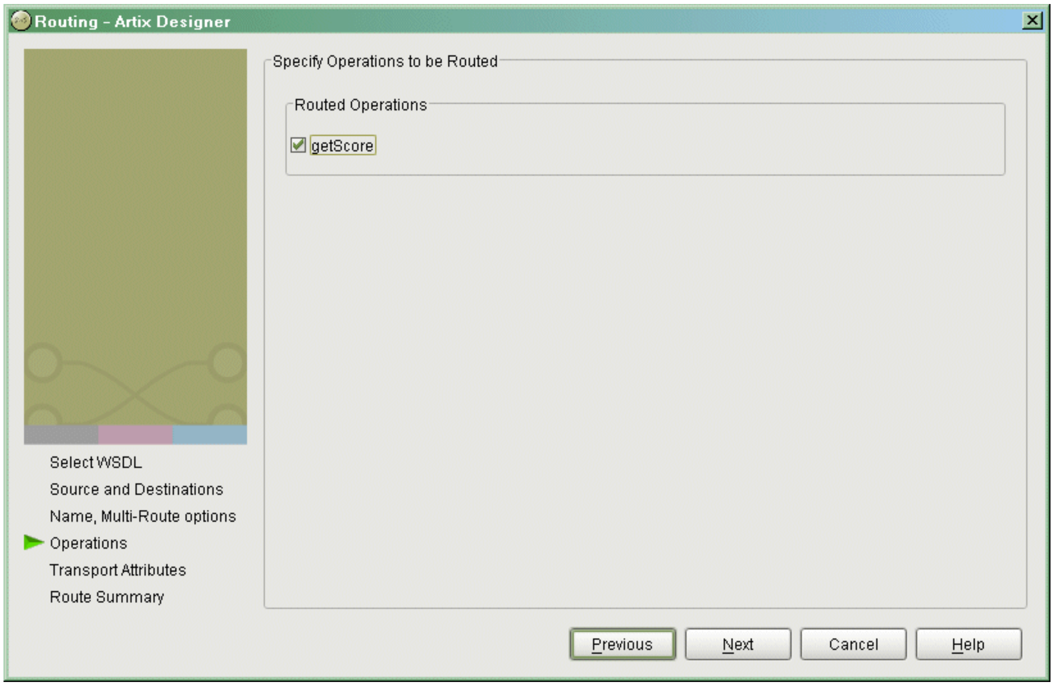
Figure 8: *Transport Attribute Routing Rules*

16. To add transport attribute based routing rules, click **Add Rule Set**.
17. The counter will automatically set itself to **0**.
18. Enter the name of the transport attribute to be used in **Name**.
19. Enter the value to be used as the test case in **Value**.
20. Click **Add** Attribute to add the attribute to the **Transport Attribute** table.
21. Once the attribute is in the table you can edit it to determine how matching attributes are compared to the value.

22. Repeat this for all the attributes you want to use in routing.

    Note: The editor has no knowledge of the valid attribute names and will allow you to enter any names and values.

23. When you are finished entering attributes, click **Next**.
24. You will see a screen like Figure 9.



Specify Operations to be Routed

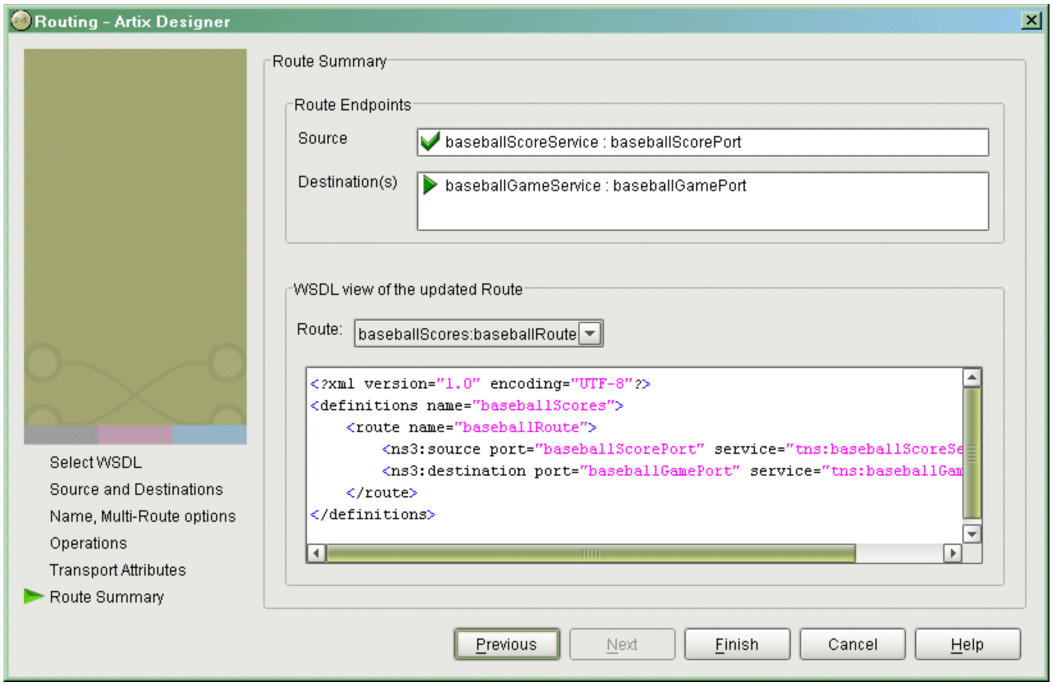
Routed Operations

[✓ getScore]

Figure 9: *Operation Routing Selection*

25. Select the desired operations to route between.
26. Click **Next**.

27. You will see a screen similar to Figure 10.



Route Summary

Route Endpoints

Source: baseballScoreService : baseballScorePort

Destination(s): baseballGameService : baseballGamePort

WSDL view of the updated Route

Route: baseballScores:baseballRoute

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="baseballScores">
    <route name="baseballRoute">
        <ns3:source port="baseballScorePort" service="tns:baseballScoreSe
        <ns3:destination port="baseballGamePort" service="tns:baseballGam
    </route>
</definitions>
```

Figure 10: *Review of Route Information*

28. Click **Finish** to create your 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 6

Using the Artix Locator Service

The Artix Locator allows Artix servers to publish their references for dynamic discovery by Artix clients.

Overview

A system with many servers cannot afford the overhead of manually propagating each servers contact information to all off the clients that might need to contact them. Given the large number of clients and the distributed nature of enterprise level deployments, the time required to accomplish this, and the room for error, are too great. Also, over time hardware upgrades, machine failures, or site reconfiguration require you to move servers and repeat the exercise of propagating the server's information to all clients.

The Artix locator isolates clients from changes in a server's contact information. The Artix contract defining how the client contacts the server contains the address for the Artix locator and it is the locator that provides the client with a reference to the server. Servers are automatically registered with the locator when they start-up.

In this Chapter

This chapter discusses the following topics:

Deploying the Locator

Overview

The Artix locator is implemented as a group of ART plug-ins. This means that any Artix application can host the locator service by loading the `service_locator` plug-in. However, it is recommended that users generate a simple Artix server that only hosts the locator service and deploy that service into there Artix environment.

In either case, the locator service requires modifications to the Artix configuration domain in which the locator is run. You also need to generate a copy of `locator.wsdl`, the contract that describes the locator service, containing the locator service's contact information.

Generating the locator service

To generate an instance of the locator service you simply need to write a simple Artix server mainline and link it with the Artix libraries. Example 22 shows an example of the locator's mainline.

Example 22: *Artix Locator Mainline*

```
include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_bus/fault_exception.h>

using namespace IT_Bus;

int main(int argc, char* argv[])
{
  try
    {
      IT_Bus::init(argc, argv, "locator_service");
      IT_Bus::run();
      IT_Bus::shutdown();
    }
  catch (IT_Bus::Exception& e)
    {
      printf("Exception occurred: %s", e.Message());
      return 1;
    }

  return 0;
}
```

The locator's `main()` only needs to initialize the Artix bus with the name of the locator's configuration scope and call `IT_Bus::run()`. The configuration scope name is the third parameter to `IT_Bus::init()`, `locator.service`. The Artix bus will load the plug-ins for the locator service.

Example 23 shows a sample makefile for building the locator service.

Example 23: *Locator Makefile*

```
IT_PRODUCT_VER = 1.2

ART_BIN_DIR=$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\bin
ART_CXX_INCLUDE_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\i
   nclude"
ART_LIB_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\lib"

CXX=cl
CXXFLAGS=-I$(ART_CXX_INCLUDE_DIR) -Zi -nologo -GR -GX -W3 -Zm250
   -MD $(EXTRA_CXXFLAGS) $(CXXLOCAL_DEFINES)

LINK=link
LDFLAGS=/DEBUG /NOLOGO
LDLIBS=/LIBPATH:$(ART_LIB_DIR) $(EXTRA_LIB_PATH) $(LINK_WITH)
   kernel32.lib ws2_32.lib advapi32.lib user32.lib

SHLIB_CXX_COMPILER_ID=          vc60
SHLIBLDFLAGS=-dll -debug -incremental:no

OBJS=$(SOURCES:.cxx=.obj)

LINK_WITH=it_bus.lib it_afc.lib it_art.lib  it_ifc.lib

SOURCES = locator.cxx
all: locator.exe

locator.exe:$(SOURCES) $(OBJS)
  if exist $@ del $@
    $(LINK) /out:$@ $(LDFLAGS) $(OBJS) $(LDLIBS)
```

The locator must be linked with the following Artix libraries:

- `it_bus.lib`
- `it_afc.lib`
- `it_art.lib`
- `it_ifc.lib`

Configuring the locator

To run the locator you need to ensure that it loads the locator service plug-in, `service_locator`. In addition, the locator must load the `soap` and `http` plug-ins as all of its communication is done using SOAP over HTTP.

In the locator's configuration scope specify the service plug-in to read the correct Artix contract for the locator by setting `plugins:locator:service_url` to point to the copy of `locator.wsdl` containing the address for this instance of the locator.

Example 24 shows the configuration scope used to start the locator.

Example 24: *Locator configuration scope*

```
locator_service
{
  plugins:locator:service_url="locator.wsdl"
  orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop", "soap", "http", "service_locator"];
};
```

For more information on Artix configuration see "Configuration" on page 27.

Generating the locator's contact information

You also need to configure the port on which the locator will run. To do this you modify `locator.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the locator service will listen. This can be either done manually for deploying the locator on a well-known fixed port, or automatically for deploying the locator on a dynamically allocated port.

To deploy the locator on a well-known fixed port, open `locator.wsdl` in any text editor and edit the `<soap:address>` entry at the bottom of the contract to specify the proper address. Example 25 shows a modified locator service contract entry. The highlighted part has been modified to point to the desired address.

Example 25: *Locator Service Address*

```
<service name="LocatorService">
  <port name="LocatorServicePort" binding="ls:LocatorServiceBinding">
    <soap:address location="http://localhost:8080/services/locator/LocatorService"/>
  </port>
</service>
```

To deploy the locator on a dynamically allocated port, configure the locator to use the copy of `locator.wsdl` shipped with Artix. Once the locator initializes the Artix bus, it will need to publish a new copy of its contract with the actual contact information. Example 26 shows how to publish the locator's contract.

Example 26: *Dynamically Located Locator Service*

```
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv,
                                   "locator_service");

// Now we write out the updated WSDL for the Locator Services

// Get the WSDL Defintions object.
IT_Bus::QName service_name("",
                           "LocatorService",
                           "http://ws.iona.com/locator");
IT_Bus::Service * service = bus->get_service(service_name);
const IT_WSDL::WSDLDefinitions & definitions =
                                 service->get_wsdl_definitions();

// Serialize the WSDL model to another wsdl file.
IT_Bus::FileOutputStream stream("active-locator.wsdl");
IT_Bus::XMLOutputStream xml_stream(stream, true);
definitions.write(xml_stream);
stream.close();

IT_Bus::run();
```

Starting the locator

Once the locator has been generated and properly configured it can be started just like any other application.

Registering a Server with the Locator

Overview

A server does not need to have its implementation changed to work with the Artix locator. All that is required is that the server be configured to load the proper plug-ins and to reference the correct locator contract.

Configuring the server

Any server that wishes to register itself with the locator must load the following plug-ins in addition to the transport and payload plug-ins it requires:

- `soap`
- `http`
- `locator_endpoint`

`locator_endpoint` allows the server to register with the running locator.

The server's configuration also needs to set `plugins:locator:wsdl_url` to point to the appropriate locator contract.

Example 27 shows the configuration scope of a server that registers with the locator service.

Example 27: *Server Configuration Scope*

```
rune_server
{
  plugins:locator:wsdl_url="locator.wsdl";
  orb_plugins = ["xmlfile_log_stream", "soap", "http", "tunnel",
    "locator_endpoint"];
 };
```

`rune_server` provides its services using SOAP over IIOP so in addition to the locator plug-ins it also loads the `tunnel` plug-in.

For more information on Artix configuration see "Configuration" on page 27.

Registration

Once a properly configured server starts up, it automatically registers with the locator specified by the contract pointed to by `plugins:locator:wsdl_url`.

You can register multiple instances of the same server with a locator. The locator will generate a pool of references for the server type. When clients make a request for a server, the locator will supply references from this pool using a round-robin algorithm.

Obtaining References from the Locator

Overview

Unlike servers, clients must be specifically written to work with the Artix locator. There are three steps a client must take to obtain a server reference from the Artix locator. They are:

1. Instantiate a proxy for the locator service.
2. Look up the desired server's endpoint using the locator service proxy.
3. Create a proxy for the desired server using the returned endpoint.

Instantiating a locator service proxy

Before a client can invoke any of the look up methods on the locator service, it must create a proxy to forward requests to the running locator. To do this the client creates an instance of `LocatorServiceClient` using the locator service's contract name, `locator.wsdl`, the locator service's QName, and the port name used in the locator service's contract, `LocatorServicePort`.

Note: For more information on Artix proxy constructors, read the *Artix C++ Programmer's Guide*.

Example 28 shows how to instantiate a locator service proxy. The parameters used to create the locator service's QName, `LocatorService` and `http://ws.iona.com/locator`, should never be modified.

Example 28: *Instantiating a Locator Service Proxy*

```
// C++
QName locator_service_name("", "LocatorService",
                           "http://ws.iona.com/locator");
locator_proxy = new LocatorServiceClient("locator.wsdl",
                                         locator_service_name,
                                         "LocatorServicePort");
```

Looking up a server's endpoint

After instantiating a locator service proxy, a client can then look up servers using the proxy's `lookup_endpoint()` method. `lookup_endpoint()` has the following signature:

```
void lookup_endpoint(lookupEndpoint input,
                     lookupEndpointResponse output);
```

`input` contains the QName of the server the client is looking up. The QName is set using the `setservice_qname()` method. The QName of the service is comprised of the service name specified in the Artix contract's `<service>` tag and the target namespace of the Artix contract.

`output` contains a reference to the server. If the locator cannot find a registered instance of the requested server, `lookup_endpoint()` returns an `endpointNotExistFault` exception.

Example 29 shows the client code to look up an instance of the widget ordering service, `orderWidgetService`.

Example 29: *Looking up a Server Using the Locator Service*

```
// C++
// Create the QName for the server
QName service_name("", "orderWidgetsService",
                     "http://widgetVendor.com/widgetOrderForm");

// Create lookup input parameter
lookupEndpoint input;
input.setservice_qname(service_name);

// The output parameter is set by lookup_endpoint
lookupEndpointResponse output;

// call lookup_endpoint on the locator proxy
locator_proxy->lookup_endpoint(input, output);
```

Creating a server proxy

The client uses the reference returned in the output parameter of `lookup_endpoint()` to instantiate a server proxy for making requests on the requested server. To instantiate the proxy use the correct proxy class for the server you have requested and pass the return value of the returned `lookupEndpointResponse`’s `getservice_endpoint()` method to the proxy class’ constructor.

Note: Because the Artix locator’s look up is only one level deep, it is possible that the original look up can return a reference to a second Artix locator. Clients running in an environment where multiple locator redirects are possible must be explicitly designed to handle this situation.

Example 30 shows the client code for creating a proxy widget server from the results of the look up performed in Example 29 on page 102.

Example 30: *Instantiate a Proxy Server*

```
// C++
orderWidgetsClient widget_proxy(output.getservice_endpoint());
```

For more information on writing Artix client code read the *Artix C++ Programmer’s Guide*.

Controlling Server Workloads

Overview

Services can request that they temporarily be taken off of the locator's list of active references. This is particularly useful for managing the workloads placed on services. When they reach a certain capacity, a service can in effect disappear from any new clients wishing to access it. When the service's workload is reduced it can then reappear and once again become available to new clients.

Procedure

To control the registered state of service you need to do the following three things:

1. Obtain a handle for the service with which you intend to work.
2. Use the obtained handle to temporarily deregister the service from the locator.
3. Use the obtained handle to reregister the service with the locator.

Get a service instance

To get an instance of a service you need to use `IT_Bus::get_service()` on a bus instance. `get_service()` takes the QName of the desired service and returns a generic service handle, `IT_Bus::Service*`.

Note: A bus instance can only return service handles for services that is activated on that particular bus.

Example 31 shows how to obtain a handle for a service from the active bus.

Example 31: *Obtaining a Service Handle*

```
//C++
// Build service QName
IT_Bus::QName service_name("", "MMService", "http://MM.com");

// Get the service handle from the active bus
IT_Bus::Service* = bus->get_service(service_name);
```

For more information on using `get_service()` see the *Artix C++ Programmer's Guide*.

Deregistering a service

To temporarily deregister a service, you use the `reached_capacity()` method of the service handle returned by the active bus. This method informs the service's endpoint manager that the service is busy and does not want to receive requests from any new clients. The endpoint manager will then contact the locator and ask to be removed from the list of available services.

Note: Clients that already have a valid reference for the service will still be able to make request on the service once it has been deregistered.

Example 32 shows how to call `reached_capacity()`.

Example 32: *Calling reached_capacity()*

```
\\ C++
\\ Service otained previously
service->reached_capacity();
```

Reregistering a service

When the service is ready to be reregistered, you use the `below_capacity()` method of the service handle used when deregistering the service. `below_capacity()` informs the endpoint manager that the service is capable of accepting requests from new clients. The endpoint manager then contacts the locator and asks to be placed on the list of available services.

Example 33 shows how to call `reached_capacity()`.

Example 33: *Calling below_capacity()*

```
\\ C++
\\ Service otained previously
service->below_capacity();
```

Fault Tolerance

Overview

Enterprise level deployments demand that applications can cleanly recover from occasional failures. The Artix locator is designed to recover from the two most common failures faced by a look-up service:

- failure of a registered end-point.
- failure of the look-up service itself.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the locator that it will no longer be available and the locator removes the endpoint from its list so it cannot give a client a reference to a dead endpoint. However, when an endpoint fails unexpectedly, it cannot notify the locator and the locator can unknowingly give a client an invalid reference causing the failure to cascade.

To mitigate the risk of passing invalid references to clients, the locator service occasionally pings all of its registered endpoints to see if they are still running. If an endpoint does not respond to a ping, the locator removes that endpoint's reference.

You can adjust the interval between locator service pings by setting the configuration variable `plugins:locator:peer_timeout`. The default setting is 4 seconds. For more information see "Configuration" on page 27.

Service failure

When the locator service fails all of the references to the registered endpoints are lost and the active endpoints are no longer registered with the locator. To ensure that the active endpoints reregister with the locator when it restarts, the endpoints, after the locator has missed its ping interval, will periodically attempt to reregister with the locator until they are successful.

You can adjust the interval at which the endpoint pings the locator by setting the configuration variable `plugins:session_endpoint_manager:peer_timout`. The default setting is 4 seconds. For more information see "Configuration" on page 27.

Using the Artix Session Manager

The Artix Session Manager provides a mechanism for managing server resources in enterprise deployments.

Overview

The Artix session manager ensures that a one to one relationship is maintained between a client and a server. When the session manager is loaded, clients are granted session tokens when requesting a service instance from the locator and the server will only accept requests that include a valid session token. All other requests will be rejected.

For example, you deploy three instances of a stock trading service into an environment with the Artix Locator and the Artix Session Manager. Each instance of the service will register with the locator when it starts up. When client A requests an instance of the service from the locator, the locator gives it a reference to instance to stock service 1 and the session manager assigns client A a session token that is only valid for stock service 1. If client 2 gets a reference to stock service 1 and makes a request, the request will be rejected as long as client A's session is valid. Client 2 will need to request a reference and valid session token from the locator. This is shown in Figure 11.

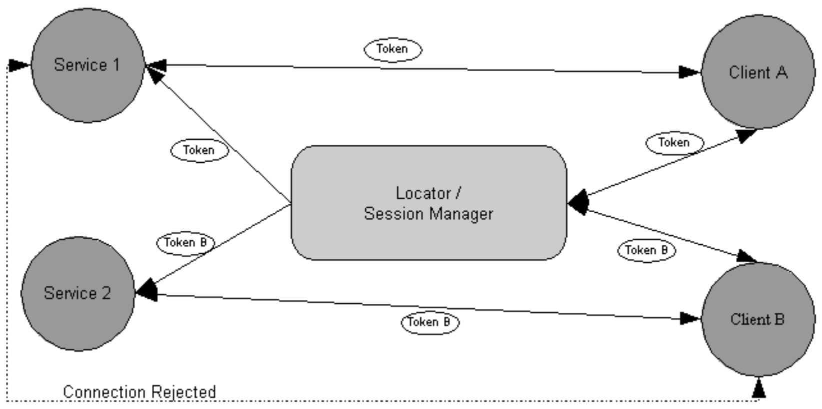


Figure 11: *Session Manager*

In this chapter

This chapter discusses the following topics:

Deploying the Session Manager

Overview

The Artix session manager is implemented as a group of ART plug-ins. This means that any Artix application can host the session manager's core functionality by loading the `session_manager_service` and `sm_simple_policy` plug-ins. However, it is recommended that users generate a simple Artix server that only hosts the session manager and deploy that server into the Artix environment.

In either case, the session manager requires modifications to the Artix configuration domain in which the locator is run. You also need to generate a copy of `session-manager.wsdl`, the contract that describes the session manager, containing the session manager's contact information.

Generating the session manager

To generate an instance of the session manager you simply need to write a simple Artix server mainline and link it with the Artix libraries. Example 34 shows an example of the session manager's mainline.

Example 34: *Artix Session Manager Mainline*

```
include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_bus/fault_exception.h>

using namespace IT_Bus;
```

Example 34: *Artix Session Manager Mainline*

```
#int main(int argc, char* argv[])
{
  try
    {
      IT_Bus::init(argc, argv, "managed_sessions");
      IT_Bus::run();
      IT_Bus::shutdown();
    }
  catch (IT_Bus::Exception& e)
    {
      printf("Exception occurred: %s", e.Message());
      return 1;
    }

  return 0;
}
```

The session manager's `main()` only needs to initialize the Artix bus with the name of the session manager's configuration scope and call `IT_Bus::run()`. The configuration scope name is third parameter to `IT_Bus::init()`, `managed_sessions`. The Artix bus will load the plug-ins for the session manager.

Example 35 shows a sample makefile for building the session manager.

Example 35: *Session Manager Makefile*

```
IT_PRODUCT_VER = 1.2

ART_BIN_DIR=$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\bin
ART_CXX_INCLUDE_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\i
   nclude"
ART_LIB_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\lib"

CXX=cl
CXXFLAGS=-I$(ART_CXX_INCLUDE_DIR) -Zi -nologo -GR -GX -W3 -Zm250
   -MD $(EXTRA_CXXFLAGS) $(CXXLOCAL_DEFINES)
```

Example 35: *Session Manager Makefile*

```
LINK=link
LDFLAGS=/DEBUG /NOLOGO
LDLIBS=/LIBPATH:$(ART_LIB_DIR) $(EXTRA_LIB_PATH) $(LINK_WITH)
   kernel32.lib ws2_32.lib advapi32.lib user32.lib

SHLIB_CXX_COMPILER_ID=          vc60
SHLIBLDFLAGS=-dll -debug -incremental:no

OBJS=$(SOURCES:.cxx=.obj)

LINK_WITH=it_bus.lib it_afc.lib it_art.lib  it_ifc.lib

SOURCES = session_manager.cxx
all: session_manager.exe

locator.exe:$(SOURCES) $(OBJS)
  if exist $@ del $@
    $(LINK) /out:$@ $(LDFLAGS) $(OBJS) $(LDLIBS)
```

The session manager must be linked with the following Artix libraries:

- `it_bus.lib`
- `it_afc.lib`
- `it_art.lib`
- `it_ifc.lib`

Configuring the session manager

To run the session manager you need to ensure that it loads the session manager service plug-in, `session_manager_service` and the session manager policy plug-in, `sm_simple_policy`. In addition, the session manager must load the `soap` and `http` plug-ins as all of its communication is done using SOAP over HTTP.

In the session manager's configuration scope you will need to specify the location for the session manager's contract by setting `plugins:session_manager_service:service_url` to point to the copy of `session-manager.wsdl` containing the contact information for this session manager.

Example 36 shows the configuration scope used to start the session manager.

Example 36: *Locator configuration scope*

```
managed_sessions
{
  orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop", "iiop", "soap", "http",
   "session_manager_service", "sm_simple_policy"];
  plugins:session_manager_service:service_url="session-namager.wsdl"
};
```

For more information on Artix configuration see "Configuration" on page 27.

Generating the locator's contact information

You also need to configure the port on which the session manager will run. To do this you modify `session-manager.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the session manager will listen. This can be either done manually for deploying the session manager on a well-known fixed port, or automatically for deploying the session manager on a dynamically allocated port.

To deploy the session manager on a well-known fixed port, open `session-manager.wsdl` in any text editor and edit the `<soap:address>` entry for the `SessionManagerService` to specify the proper address. Example 37 shows a modified session manager contract entry. The highlighted part has been modified to point to the desired address.

Example 37: *Session Manager Address*

```
<service name="SessionManagerService">
  <port name="SessionManagerPort" binding="sm:SessionManagerBinding">
    <soap:address
   location="http://localhost:8080/services/sessionManagement/sessionManagerService"/>
  </port>
</service>
```

To deploy the session manager on a dynamically allocated port, configure the session manager to use the copy of `session-manager.wsdl` shipped with Artix. Once the session manager initializes the Artix bus, it will need to publish a new copy of its contract with the actual contact information. Example 38 shows how to publish the session manager's contract.

Example 38: *Dynamically Located Locator Service*

```
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv,
                                   "managed-sessions");

// Now we write out the updated WSDL for the session manager

// Get the WSDL Defintions object.
IT_Bus::QName service_name("",
                           "SessionManagerService",
                          "http://ws.iona.com/session-manager");
IT_Bus::Service * service = bus->get_service(service_name);
const IT_WSDL::WSDLDefinitions & definitions =
                                 service->get_wsdl_definitions();

// Serialize the WSDL model to another wsdl file.
IT_Bus::FileOutputStream stream("active-smservice.wsdl");
IT_Bus::XMLOutputStream xml_stream(stream, true);
definitions.write(xml_stream);
stream.close();

IT_Bus::run();
```

Starting the session manager

Once the session manager has been generated and properly configured it can be started just like any other application. The only caveat is that the session manager must be started before any servers that need to register with it.

Registering a Server with the Session Manager

Overview

Services that wish to be managed by the session manager must register with a running session manager. To do this the servers instantiating these services must load the session manager endpoint plug-in and properly configure themselves. They do not require any special application code.

Once registered with a session manager, the services will only accept requests containing a valid session header. All clients wishing to access the services must be written to support session managed services.

Configuring the server

Any server that wishes to be managed by the session manager must load the following plug-ins in addition to the transport and payload plug-ins it requires:

- `soap`
- `http`
- `session_endpoint_manager`

`session_endpoint_manager` allows the server to register with a running session manager.

The server's configuration also needs to set the following configuration variables:

plugins:session_endpoint_manager:wsdl_url points to the contract describing the contact information for the session manager that will be managing the server.

plugins:session_endpoint_manager:endpoint_manager_url points to the contract describing the contact information for the endpoint manager for this server. This enables the session manager to contact the service to with updated state information.

plugins:session_endpoint_manager:default_group specifies the default group name for the services instantiated by the server.

Note: While the session manager does not require it, it is recommended that all services in a group implement the same interface.

Example 39 shows the configuration scope of a server that is managed by the session manager.

Example 39: *Server Configuration Scope*

```
qajaq_server
{
  orb_plugins = ["xmlfile_log_stream", "soap", "http", "fixed", "session_endpoint_manager"];
  plugins:session_endpoint_manager:wsdl_url="session-manager-service.wsdl";
  plugins:session_endpoint_manager:endpoint_manager_url="session-manager-endpoint.wsdl";
  plugins:session_endpoint_manager:deafult_group="qajaq_group";
 };
```

A server loaded into the `qajaq_server` configuration scope will be managed by the session manager at the location specified in `session-manager-service.wsdl`, its endpoint manager will come up at the address specified in `session-manager-endpoint.wsdl`, and by default all services instantiated by the server will belong to the session manager group `qajaq_group`.

For more information on Artix configuration see "Configuration" on page 27.

You also need to configure the port on which the endpoint manager will run. To do this you modify `session-manager.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the endpoint manager will be available. Using any text editor, open `session-manager.wsdl` and edit the `<soap:address>` entry for the `SessionEndpointManagerService` to specify the proper address. Example 40 shows a modified session manager contract entry. The highlighted part has been modified to point to the desired address.

Example 40: *Endpoint Manager Address*

```
<service name="SessionEndpointManagerService">
  <port name="SessionEndpointManagerPort" binding="sm:SessionEndpointManagerBinding">
    <soap:address
   location="http://localhost:8080/services/sessionManagement/sessionEndpointManager"/>
  </port>
</service>
```

In the server's configuration scope specify the endpoint manager plug-in to read the correct Artix contract for the endpoint manager by setting `plugins:session_endpoint_mamanger:endpoint_manager_url` to point to the copy of `session-manager.wsdl` containing the address for this instance of the endpoint manager.

Registration

Once a properly configured server starts up, it automatically registers with the session manager specified by the contract pointed to by `plugins:session_endpoint_manager:wsdl_url`.

You can register multiple instances of the same server with a session manager. The session manager generates a pool of references for the server type and associate them by their group. Clients are given a list of all available endpoints in a given group and can request a session from the pool.

Working with Sessions

Overview

Clients that wish to make requests from session managed services must be designed explicitly to interact with the Artix session manager and pass session headers to the session managed services.

There are eight steps a client takes when making requests on a session managed service. They are:

1. Instantiate a proxy for the session management service.
2. Start a session for the desired service's group using the session manager proxy.
3. Obtain the list of endpoints available in the group.
4. Create a service proxy from one of the endpoints in the group.
5. Build a session header to pass to the service.
6. Invoke requests on the endpoint using the proxy.
7. Renew the session as needed.
8. End the session using the session manager proxy when finished with the endpoint.

Instantiating a session manager proxy

Before a client can request a session from the session manager, it must create a proxy to forward requests to the running session manager. To do this the client creates an instance of `SessionManagerClient` using the locator service's contract name, `session-manager.wsdl`, the session manager's QName, and the port name used in the locator service's contract, `SessionManagerPort`.

Note: For more information on Artix proxy constructors, read the *Artix C++ Programmer's Guide*.

Example 41 shows how to instantiate a session manager proxy. The parameters used to create the session manager's QName, `SessionManager` and `http://ws.iona.com/session_manager`, should never be modified.

Example 41: *Instantiating a Session Manager Proxy*

```
// C++
QName session_manager_name("", "SessionManager",
                           "http://ws.iona.com/session_manager");
locator_proxy = new LocatorServiceClient("session_manager.wsdl",
                                         session_manager_name,
                                         "SessionManagerPort");
```

Start a session

After instantiating a session manager proxy, a client can then start a session for the desired service's group using the session manager's `begin_session()` method. `begin_session()` has the following signature:

```
void begin_session(IT_Bus_Services::BeginSession input,
                   IT_Bus_Services::BeginSessionResponse output);
```

`input` contains the name of the group containing the desired service and the duration that the session is valid for before it needs to be renewed. The group name is set using the `setendpoint_group()` method. The group name can be any valid string and corresponds to the default group name set in the service's configuration scope as described in "Configuring the server" on page 114.

The session duration is set using the `setprefered_renew_timeout()` method. The duration is specified in seconds. If the specified duration is less than the value specified by the service's `min_session_timeout` configuration setting, it will be set to the configured minimum value. If the specified duration is higher than the value specified by the service's `max_session_timeout` configuration setting, it will be set the configured max value. For more information see "Configuration" on page 27.

`output` contains the information needed to use the session.

Once a session is returned in `output`, you will need to extract the session ID to work with the session. This is done using `IT_Bus_Services::ActiveSession::getsession_id()`. `getsession_id()` returns the session ID as an `IT_Bus_Services::SessionID`.

Example 42 shows the client code to begin a session for `qajaq_group`.

Example 42: *Beginning a Session*

```
// C++
IT_Bus_Services::BeginSession begin_session_request;
IT_Bus_Services::BeginSessionResponse begin_session_response;

// set the group to request
begin_session_request.setendpoint_group("qajaq_group");
// set session renewal interval to 10 mins
begin_session_request.setpreferred_renew_timeout(600);

session_mgr.begin_session(begin_session_request,
                          begin_session_response);

IT_Bus_Services::SessionId session;
session =
   begin_session_response.getsession_info().getsession_id();
```

Get a list of endpoints in the group

The session manager hands out sessions for a group of services, so in order to get an individual endpoint upon which to make requests a client needs to get a list of the endpoints in the session's group. The session manager proxy's `get_all_endpoints()` method returns a list of all endpoints registered to the specified group. `get_all_endpoints()` has the following signature:

```
void get_all_endpoints(IT_Bus_Services::GetAllEndpoints request,
        IT_Bus_Services::GetAllEndpointsResponse response)
```

`request` contains the session ID for which you are requesting endpoints. Set the session ID using the `setsession_id()` method on `request` with the session ID returned from the session manager.

`response` contains the list of endpoints returned from `get_all_endpoints()`. If the group has no endpoints, `response` will be empty.

Example 43 shows how to get the list of endpoints for a group.

Example 43: *Retrieving the List of Endpoints in a Group*

```
//C++
IT_Bus_Services::GetAllEndpoints request;
IT_Bus_Services::GetAllEndpointsResponse response;

// group session initialized above.
get_all_endpoints_request.setsession_id(session);

session_mgr.get_all_endpoints(request, response);
```

Create a proxy for the requested service

The client can use any of the endpoints returned by `get_all_endpoints()` to instantiate a server proxy for making requests on the requested server. To instantiate the proxy, you first need to narrow down the list returned endpoints to the desired one. `GetAllEnpointsResponse` contains an array of references to active endpoints that can be retrieved using `GetAllEndpointsResponse`'s `getendpoints()` method. You can use simple indexing to get one of the references. For example, to use the first endpoint you would use the following:

```
response.getendpoints()[0]
```

Because the session manager simply returns the endpoints in the order the services registered with the session manager, the clients must be responsible for circulating through the list or else they will all make requests on only one service in the group. Also, because the session manager does not force all members of a group to implement the same interface, you may want to have your clients check each endpoint to see if it implements the correct interface by checking the reference's service name as shown in Example 44.

Example 44: *Checking the Endpoint for its Interface*

```
//C++
IT_Bus::Reference endpoint = response.getendpoints()[0];
if (endpoint.get_service_name() ==
    QName("", "QajaqService", "http://qajaqs.com"))
  {
  // instantiate a QajaqService using endpoint
  }
else
  {
  // do something else
  }
```

Example 45 shows the client code for creating a proxy `qajaq` server from a group endpoint.

Example 45: *Instantiate a Proxy Server*

```
// C++
QajaqClient qajaq_proxy(response.getendpoints()[0]);
```

Create a session header

Services that are being managed by the session manager will only accept requests that include a valid session header. The session header information is passed to the server as part of the proxy's input message attributes. Creating the session header and putting into the input message attributes takes three steps:

1. Set the proxy to use input message attributes.
2. Get a handle to the proxy's input message attributes.
3. Set the session information into the input message attributes.

Setting the proxy to use input message attributes

Artix client proxies all support a helper method, `get_port()`, that provides access to the port information used by the client to connect the server. One of an Artix proxy's port properties is `use_input_message_attributes`.

Setting this property to `true` tells the bus to endure the input message attributes are propagated through to the server. Example 46 shows how to set the client proxy port's `use_input_message_attributes` property to `true`.

Example 46: *Use Input Message Attributes*

```
//C++
// Get the proxy's port
IT_Bus::Port proxy_port = qajaq_proxy.get_port();

// set the port property
proxy_port.use_input_attributes(true);
```

Getting a handle to the input message attributes

A pointer to the proxy port's input message attributes is returned by the port's `get_input_message_attributes()` method. Example 47 shows how to get a handle to the input message attributes.

Example 47: *Getting the Input Message Attributes*

```
MessageAttributes& input_attributes =
    proxy_port().get_input_message_attributes();
```

Setting the session information into the input message attributes

There are two attributes that need to be set to include the proper session information in the input message:

SessionName specifies the name the session manager has given this session. The session manager endpoints in the group will also be given this name to validate session header's against. The session name is returned by invoking `getname()` of the session ID of the active session.

SessionGroup specifies the group name for which the session is valid. The session endpoints also use to ensure that the session is for the correct group. The session group is returned by invoking `getendpoint_group()` on the session ID of the active session.

The input message attributes are set using the message attribute handle's `set_string()` method. `set_string()` takes two attributes. The first is a string specifying the name of the attribute being set. The second is the value to be set for the attribute. Example 48 shows how to set the session information in to the input message attributes.

Example 48: *Setting the Input Message Attributes*

```
// C++
input_attributes.set_string("SessionName", session.getname());
input_attributes.set_string("SessionGroup",
                            session.getendpoint_group());
```

Make requests on service proxy

Once the session information is added to the proxy's port information, the client can invoke operations on the client as it would a non-managed server. If the endpoint rejects the request because the client's session is not valid, an exception is raised.

Renewing a session

If a client is going to use a session for a longer than the duration requested when the session was granted, the client will need to renew its session or the session will timeout. A session is renewed using the session manager proxy's `renew_session()` method. `renew_session()` has the following signature:

```
void renew_session(IT_Bus_Services::RenewSession params,
                   IT_Bus_Services::RenewSessionResponse renewed);
```

`params` contains the session ID of the session being renewed and the duration, in seconds, of the renewal. The session ID is set using `params`' `setsession_id()` method. The renewal duration is set using `params`' `setrenew_timeout()` method.

If the renewal is successful, `renewed` will return containing the duration of the renewal. The returned duration may be different if the requested renewal duration was outside of the configured range for session timeouts.

If the renewal is unsuccessful, an `IT_Bus_Services::renewSessionFaultException` is raised.

Example 49 shows how to end a session.

Example 49: *Ending a Session*

```
//C++
IT_Bus_Services::RenewSession params;
IT_Bus_Services::RenewSessionResponse renewed;
params.setsession_id(session);
parames.setrenewal_timeout(600);
try
{
  session_mgr.renew_session(params, renewed);
}
catch (IT_Bus_Services::renewSessionFaultException)
{
  // handle the exception
}
```

End the session

When a client is finished with a session managed service, it should explicitly end its session. This will ensure that the session will be freed up immediately. A session is ended using the session manager proxy's `end_session()` method. `end_session()` has the following signature:

```
void end_session(IT_Bus_Services::EndSession params);
```

`params` contains the session ID of the session being ended. The session ID is set using `params`' `setsession_id()` method.

Example 50 shows how to end a session.

Example 50: *Ending a Session*

```
//C++
IT_Bus_Services::EndSession params;
params.setsession_id(session);
session_mgr.end_session(params);
```

For more information on writing Artix client code read the *Artix C++ Programmer's Guide*.

Fault Tolerance

Overview

Enterprise level deployments demand that applications can cleanly recover from occasional failures. The Artix session is designed to recover from the two most common failures:

- failure of a registered endpoint.
- failure of the session manager itself.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the session manager that it will no longer be available and the session manager removes the endpoint from its list so it cannot give a client a reference to a dead endpoint. However, when an endpoint fails unexpectedly, it cannot notify the session manager and the session manager can unknowingly give a client an invalid reference causing the failure to cascade.

To mitigate the risk of passing invalid references to clients, the session manager occasionally pings all of its registered endpoints to see if they are still running. If an endpoint does not respond to a ping, the session manager removes that endpoint's reference.

You can adjust the interval between session manager pings by setting the configuration variable `plugins:session_manager:peer_timeout`. The default setting is 4 seconds. For more information see "Configuration" on page 27.

Service failure

When the session manager fails all of the references to the registered endpoints are lost and the active endpoints are no longer be registered. To ensure that the active endpoints reregister with the session manager when it restarts, the endpoints, after the session manager has missed its ping interval, will periodically attempt to reregister with the session manager until they are successful.

You can adjust the interval between the endpoint's pings of the session manager by setting the configuration variable `plugins:session_endpoint_manager:peer_timout`. The default setting is 4 seconds. For more information see "Configuration" on page 27.

CHAPTER 8

Artix Logging and SNMP Support

This chapter describes various Artix logging approaches, including Artix support for SNMP (Simple Network Management Protocol) and integration with third-party SNMP management tools.

In this chapter

This chapter includes the following sections:

Artix Logging

Overview

Artix provides the following `IT_Logging::logstream` plug-ins: the `xmlfile_logstream` and `snmp_logstream`. In addition, IONA Application Server Platform logging features such as `local_logstream`. are provided.

For information on configuring these plugins see "Configuration" on page 27.

Using Trace Macros

Artix Trace Macros

In using Trace macros, the most important concept is the trace level. Trace level is an enum, defined in `it_bus/logging_support`, that lets you filter events:

```
const IT_TraceLevel IT_TRACE_FATAL = 64;           //FATAL
const IT_TraceLevel IT_TRACE_ERROR = 32;           //ERROR
const IT_TraceLevel IT_TRACE_WARNING = 16;         //WARNING
const IT_TraceLevel IT_TRACE = 4;                   //INFO_HIGH
const IT_TraceLevel IT_TRACE_BUFFER = 2;           //INFO_MED
const IT_TraceLevel IT_TRACE_METHODS = 1;          //INFO_LOW
const IT_TraceLevel IT_TRACE_METHODS_INTERNAL = 1; //INFO_LOW
```

The simplest trace statement emits a constant string at level `IT_TRACE`. For example:

```
TRACELOG("Hello world");
```

Several versions of the macro allow using a C `printf` format string, and passing in some arguments. Because you cannot have variable argument lists for macros, there are several defined according to how many arguments are allowed:

```
TRACELOG1("My name is: %s", "Slim Shady");
TRACELOG2("At state number %d, this happened: %s", 44, "connection failure");
```

Both the zero argument and the multi argument versions have a set that allows a trace level to be passed in, instead of level `IT_TRACE`. For example:

```
TRACELOG_WITH_LEVEL(IT_METHODS, "MyClass::MyClass()");
TRACELOG_WITH_LEVEL1(IT_TRACE_METHODS_INTERNAL, "Value of my_name_field was %s", my_name_field);
```

If you must create your own output using `iostreams` or another expensive process that isn't supported by the macro, you use the trace guard block, so that the trace level test will prevent your trace creation code from running when it will not produce output. For example:

```
BEGIN_TRACE(IT_TRACE)
        String trace_message = "data elements: ";
        for(i = 0; i < data_count; i++)
        {
                trace_message = trace_message + data_item[i] + "
   ";
        }
        TRACELOG(trace_message.c_str());
END_TRACE
```

To create binary output (for instance, a hex dump of the buffer), use `TRACELOGBUFFER`. For example:

```
TRACELOGBUFFER(vvMQMessageData, vvMQMessageData.GetSize())
```

If the trace statement issues at a level less than or equal to the process trace level, then the entry is written to disk. The default log file name is `it_bus.log`.

Application Server Platform Trace Macros

<orbix\logging_support.h> defines ASP-style logging macros.

IT_LOG_MESSAGE Macros

IT_LOG_MESSAGE() Macro

```
// C++
#define IT_LOG_MESSAGE( \
    event_log,  \
    subsystem, \
    id, \
    severity, \
    desc \
) ...
```

A macro to use for reporting a log message.

Parameters

event_log	The log (EventLog) where the message is to be reported.
subsystem	The SubsystemId.
id	The EventId.
severity	The EventPriority.
desc	A string description of the event.

Examples

Here is a simple example of usage:

```
...
IT_LOG_MESSAGE(
    event_log,
    IT_IIOP_Logging::SUBSYSTEM,
    IT_IIOP_Logging::SOCKET_CREATE_FAILED,
    IT_Logging::LOG_ERROR,
    SOCKET_CREATE_FAILED_MSG
);
```

IT_LOG_MESSAGE_1() Macro

```
// C++
#define IT_LOG_MESSAGE_1( \
    event_log, \
    subsystem, \
    id, \
    severity,  \
    desc, \
    param0 \
) ...
```

A macro to use for reporting a log message with one event parameter.

Parameters

`event_log`	The log (`EventLog`) where the message is to be reported.
`subsystem`	The `SubsystemId`.
`id`	The `EventId`.
`severity`	The `EventPriority`.
`desc`	A string description of the event.
`param0`	A single parameter for an `EventParameters` sequence.

In addition, the `IT_LOG_MESSAGE_2()`, `IT_LOG_MESSAGE_3()`, `IT_LOG_MESSAGE_4()`, and `IT_LOG_MESSAGE_5()` macros, are provided for reporting log messages with two, three, four, and five parameters, respectively.

Logging to a File

Using the SNMP Logging Plug-in

SNMP

The Artix SNMP LogStream plug-in uses the open source library `net-snmp` (v.5.0.7) to emit SNMPv1/v2 traps. For more information on this implementation, see http://sourceforge.net/projects/net-snmp/. To obtain a freeware SNMP Trap Receiver, visit http://www.ncomtech.com.

Simple Network Management Protocol (SNMP) is the Internet standard protocol for managing nodes on an IP network. SNMP can be used to manage and monitor all sorts of equipment (for example, network servers, routers, bridges, and hubs).

the Artix Management Information Base (MIB)

A *MIB file* is a database of objects that can be managed using SNMP. It has a hierarchical structure, similar to a DOS or UNIX directory tree. It contains both pre-defined values and values that can be customized. The Artix MIB is shown below:

Example 51: *Artix MIB*

```
IONA-ARTIX-MIB DEFINITIONS  ::= BEGIN

 IMPORTS
        MODULE-IDENTITY, OBJECT-TYPE,
       Integer32, Counter32,
       Unsigned32,
       NOTIFICATION-TYPE              FROM   SNMPv2-SMI
       DisplayString                  FROM   RFC1213-MIB
;

-- v2 s/current/current


 iona OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) 3027 }

 ionaMib MODULE-IDENTITY
 LAST-UPDATED "200303210000Z"

 ORGANIZATION "IONA Technologies PLC"
 CONTACT-INFO
             "
                Corporate Headquarters
                Dublin Office
                The IONA Building
                Shelbourne Road
                Ballsbridge
                Dublin 4 Ireland
                Phone: 353-1-662-5255
                Fax: 353-1-662-5244

                US Headquarters
                Waltham Office
                200 West Street 4th Floor
                Waltham, MA 02451
                Phone: 781-902-8000
                Fax: 781-902-8001

                Asia-Pacific Headquarters
                IONA Technologies Japan, Ltd
                Akasaka Sanchome Bldg.
                7F 3-21-16 Akasaka, Minato-ku,
                Tokyo, Japan 107-0052
                Tel: +81 3 3560 5611
                Fax: +81 3 3560 5612
```

Example 51: *Artix MIB*

```
              E-mail: support@iona.com
             "
 DESCRIPTION
        "This MIB module defines the objects used and format of SNMP traps that are generated
         from the Event Log for Artix based systems from IONA Technologies"

 ::= { iona 1 }



--                      iona(3027)

--                        |
--                     ionaMib(1)
--                        |
--              ________________________________________
--              |                 |                    |
--           orbix3(2)       IONAAdmin (3)          Artix (4)
-                                                     |
--                                              --------------------
--                                              |                  |
--                               ArtixEventLogMibObjects(0)  ArtixEventLogMibTraps (1)
--                                   |                                   |
--              ----------------------------------------      -----------------------
--                              |- eventSource (1)                    |- ArtixbaseTrapDef (1)
--                              |- eventId (2)
--                              |- eventPriority (3)
--                              |- timeStamp (4)
--                              |- eventDescription (5)



 Artix                          OBJECT IDENTIFIER  ::= { ionaMib 4 }
 ArtixEventLogMibObjects        OBJECT IDENTIFIER  ::= { Artix 0 }
 ArtixEventLogMibTraps          OBJECT IDENTIFIER  ::= { Artix 1 }
 ArtixBaseTrapDef               OBJECT IDENTIFIER  ::= { ArtixEventLogMibTraps 1 }


-- MIB variables used as varbinds
 eventSource         OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "The component or subsystem which generated the event."
```

Example 51: *Artix MIB*

```
    ::= { ArtixEventLogMibObjects 1 }

eventId        OBJECT-TYPE
   SYNTAX      INTEGER
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
       "The event id for the subsystem which generated the event."

   ::= { ArtixEventLogMibObjects 2 }

eventPriority      OBJECT-TYPE
   SYNTAX          INTEGER
   MAX-ACCESS      not-accessible
   STATUS          current
   DESCRIPTION
       "The severity level of this event.  This maps to IT_Logging::EventPriority types.  All
        priority types map to four general types: INFO (I), WARN (W), ERROR (E), FATAL_ERROR (F)"

   ::= { ArtixEventLogMibObjects 3 }

timeStamp       OBJECT-TYPE
   SYNTAX      DisplayString (SIZE(0..255))
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
       "The time when this event occurred."

   ::= { ArtixEventLogMibObjects 4 }

eventDescription       OBJECT-TYPE
   SYNTAX           DisplayString (SIZE(0..255))
   MAX-ACCESS       not-accessible
   STATUS           current
   DESCRIPTION
       "The component/application description data included with event."

   ::= { ArtixEventLogMibObjects 5 }

-- SNMPv1 TRAP definitions
-- ArtixEventLogBaseTraps   TRAP-TYPE
--    OBJECTS {
--       eventSource,
--       eventId,
--       eventPriority,
```

Example 51: *Artix MIB*

```
--        timestamp,
--        eventDescription
--     }

--     STATUS current
--     ENTERPRISE iona
--     VARIABLES { ArtixEventLogMibObjects }
--     DESCRIPTION  "The generic trap generated from an Artix Event Log."
--     ::= { ArtixBaseTrapDef 1 }

-- SNMPv2 Notification type

 ArtixEventLogNotif   NOTIFICATION-TYPE
    OBJECTS {
       eventSource,
       eventId,
       eventPriority,
       timestamp,
       eventDescription
    }

    STATUS current
    ENTERPRISE iona
    DESCRIPTION  "The generic trap generated from an Artix Event Log."
    ::= { ArtixBaseTrapDef 1 }

END
```

IONA SNMP integration

Events received from various Artix components are converted into SNMP management information. This information is sent to designated hosts as SNMP traps, which can be received by any SNMP managers listening on the hosts. In this way, Artix enables SNMP managers to monitor Artix-based systems.

Artix supports SNMP version 1 and 2 traps only.

Artix provides a logstream plug-in called `snmp_log_stream`. The shlib name of the SNMP plug-in found in the `artix.cfg` file is:

```
plugins:snmp_log_stream:shlib_name = "it_snmp"
```

The SNMP plug-in has five configuration variables, whose defaults can be overridden by the user. The availability of these variables is subject to change. The variables and defaults are:

```
plugins:snmp_log_stream:community = "public";

plugins:snmp_log_stream:server    = "localhost";

plugins:snmp_log_stream:port      = "162";

plugins:snmp_log_stream:trap_type = "6";

plugins:snmp_log_stream:oid       = "<your IANA number in dotted decimal notation>"
```

The last plugin described, `oid`, is the Enterprise Object Identifier. This identifier is assigned to specific enterprises by the Internet Assigned Numbers Authority (IANA). The first six numbers correspond to the prefix: "iso.org.dod.internet.private.enterprise" (1.3.6.1.4.1). Each enterprise is assigned a unique number, and can provide additional numbers to further specify the enterprise and product. For example, the `oid` for IONA is 3027. IONA has added "1.4.1.0" for Artix. Thus the complete OID for IONA's Artix is "1.3.6.1.4.1.3027.1.4.1.0". To find the number for your enterprise, visit the IANA website at http://www.iana.org.

The SNMP plug-in implements the `IT_Logging::LogStream` interface and hence, acts like the `local_log_stream` plug-in.

Using the XML Logging Plug-in

Using the XML Logging Plug-in

You can modify your event log filters to enable or disable Artix tracing.

The out of the box setting for `event_log:filters` is `["*=FATAL+ERROR"]`.

So, for example, to cause transport buffer events to be shown, update the event_log:filters to includel `INFO_MED`:

```
event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_MED"];
```

The following causes typical trace statement output:

```
event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_HI"];
```

In addition, you can:

- add `xmlfile_log_stream` to the `orb_plugins` list
- update the filename variable (default is it_bus.log):

  ```
  plugins:xmlfile_log_stream:filename = "artix_logfile.xml";
  ```

- modify the size element (default is 2MB):

  ```
  plugins:xmlfile_log_stream:max_file_size = "100000";
  ```

- add optional element (default is false):

  ```
  plugins:xmlfile_log_stream:use_pid = "false";
  ```

The Artix logging output from the TRACE macros now goes to the event log, so `local_log_stream`, `xmlfil_log_stream` or SNMP_log_stream can be used.

logging_support.h

Example 52: *Artix logging_support.h*

```
#if !defined(_IT_BUS_LOGGING_)
#define _IT_BUS_LOGGING_
#include <stdio.h>
#include <stdarg.h>

#include <it_bus/API_Defines.h>

#define MAX_STACK_ALLOCATION 256
#define MAX_TRACE_SIZE 16384

typedef IT_UShort IT_TraceLevel;

//these are now equal to ART logging values, these are just for backward compatibility
                                                    //value to put in event_log:filters
const IT_TraceLevel IT_TRACE_FATAL = 64;            //FATAL
const IT_TraceLevel IT_TRACE_ERROR = 32;            //ERROR
const IT_TraceLevel IT_TRACE_WARNING = 16;          //WARNING
const IT_TraceLevel IT_TRACE = 4;                   //INFO_HIGH
const IT_TraceLevel IT_TRACE_BUFFER = 2;            //INFO_MED
const IT_TraceLevel IT_TRACE_METHODS = 1;           //INFO_LOW
const IT_TraceLevel IT_TRACE_METHODS_INTERNAL = 1;  //INFO_LOW

extern IT_AFC_API IT_TraceLevel g_log_filter;

namespace CORBA
{
class ORB;
};

namespace IT_Logging
{
    class EventLog;
}
```

Example 52: *Artix logging_support.h*

```
extern "C"
{
    void IT_AFC_API set_global_log_filter(IT_TraceLevel trace_level);
    void IT_AFC_API set_logging_default_ORB(CORBA::ORB* orb);

    void IT_AFC_API write_log_record(IT_Logging::EventLog* event_log, IT_TraceLevel trace_level,
   const char* description, ...);
    void IT_AFC_API write_log_record_with_CDATA(IT_Logging::EventLog* event_log, IT_TraceLevel
   trace_level, const char* description, const char* data_buffer, long buffer_size);
    void IT_AFC_API write_log_record_with_binary(IT_Logging::EventLog* event_log, IT_TraceLevel
   trace_level, const char* description, const char* data_buffer, long buffer_size);
}

//These are for writing data buffers
//binary buffers are written in a hex dump format.
//to see output from these, include INFO_MED in your event_log:filters
#define IT_LOG_BUFFER(event_log, Entry, Length) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
       write_log_record_with_binary(event_log, IT_TRACE_BUFFER, "Buffer Output", Entry, Length);
   \
    }

#define IT_LOG_CDATA(event_log, description, Entry) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(event_log, IT_TRACE_BUFFER, description, Entry, 0); \
    }

#define IT_LOG_CDATA_SIZE(event_log, description, Entry, Size) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(event_log, IT_TRACE_BUFFER, description, Entry, Size); \
    }

#define IT_LOG_CDATA_BINARY_BUFFER(event_log, description, bbData) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_binary(event_log, IT_TRACE_BUFFER, description,
   bbData.get_const_pointer(), bbData.get_size()); \
    }
```

Example 52: *Artix logging_support.h*

```
//these are used for controlled tracing operations.  description is a printf format string
//they allow specifying the trace level so callers can control visibility
#define IT_LOG_GUARDED0(event_log, trace_level, description) \
    if ((g_log_filter & trace_level) != 0) \
        write_log_record(event_log, trace_level, description);

#define IT_LOG_GUARDED(event_log, trace_level, description) \
    IT_LOG_GUARDED0(event_log, trace_level, description)

#define IT_LOG_GUARDED1(event_log, trace_level, description, Arg1) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1); \
    }

#define IT_LOG_GUARDED2(event_log, trace_level, description, Arg1, Arg2) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1, Arg2); \
    }

#define IT_LOG_GUARDED3(event_log, trace_level, description, Arg1, Arg2, Arg3) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1, Arg2, Arg3); \
    }

#define IT_LOG_GUARDED4(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4); \
    }

#define IT_LOG_GUARDED5(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5) \
    if ((g_log_filter & trace_level) != 0) \
    { \
           write_log_record(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5); \
    }
```

Example 52: *Artix logging_support.h*

```
//these are used to guard a code block from executing when the purpose of the code
//block is solely for formatting a trace statement.  It prevents the code from
//executing when the trace_level is filtered out and wouldn't be used anyway.
#define BEGIN_TRACE(trace_level)                                        \
    if ((g_log_filter & trace_level) != 0)                               \
    {

#define END_TRACE                                                                   \
    }


//all the macros that follow are just short hand for the previous ones, but they
//default the event_log to 0, which uses the first one that was loaded (usually
//the only one unless you are using multiple orb names in your cfg file

//These are for writing data buffers
//binary buffers are written in a hex dump format.
//to see output from these, include INFO_MED in your event_log:filters
#define TRACELOGBUFFER(Entry, Length) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_binary(0, IT_TRACE_BUFFER, "Buffer Output", Entry, Length); \
    }

#define TRACELOG_CDATA(description, Entry) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(0, IT_TRACE_BUFFER, description, Entry, 0); \
    }

#define TRACELOG_CDATA_SIZE(description, Entry, Size) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(0, IT_TRACE_BUFFER, description, Entry, Size); \
    }

#define TRACELOG_CDATA_BINARY_BUFFER(description, bbData) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
       write_log_record_with_binary(0, IT_TRACE_BUFFER, description, bbData.get_const_pointer(), 
   bbData.get_size()); \
    }
```

Example 52: *Artix logging_support.h*

```
//These are used for method level tracing
//to see output from these, include INFO_LOW in your event_log:filters
#define BEGIN_INTERNAL_METHOD(Name) \
    const char *FuncName = Name; \
    if ((g_log_filter & IT_TRACE_METHODS_INTERNAL) != 0) \
        write_log_record(0, IT_TRACE_METHODS_INTERNAL, FuncName);

#define END_INTERNAL_METHOD

#define BEGIN_METHOD(Name) \
    const char *FuncName = Name; \
    if ((g_log_filter & IT_TRACE_METHODS_INTERNAL) != 0) \
        write_log_record(0, IT_TRACE_METHODS, FuncName);

#define END_METHOD


//these are used for controlled tracing operations.  description is a printf format string
//they allow specifying the trace level so callers can control visibility
#define TRACELOG_WITH_LEVEL0(trace_level, description) \
    IT_LOG_GUARDED(0, trace_level, description)

#define TRACELOG_WITH_LEVEL(trace_level, description) \
    IT_LOG_GUARDED(0, trace_level, description)

#define TRACELOG_WITH_LEVEL1(trace_level, description, Arg1) \
    IT_LOG_GUARDED1(0, trace_level, description, Arg1)

#define TRACELOG_WITH_LEVEL2(trace_level, description, Arg1, Arg2) \
    IT_LOG_GUARDED2(0, trace_level, description, Arg1, Arg2)

#define TRACELOG_WITH_LEVEL3(trace_level, description, Arg1, Arg2, Arg3) \
    IT_LOG_GUARDED3(0, trace_level, description, Arg1, Arg2, Arg3)

#define TRACELOG_WITH_LEVEL4(trace_level, description, Arg1, Arg2, Arg3, Arg4) \
    IT_LOG_GUARDED4(0, trace_level, description, Arg1, Arg2, Arg3, Arg4)

#define TRACELOG_WITH_LEVEL5(trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5) \
    IT_LOG_GUARDED5(0, trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5)
```

Example 52: *Artix logging_support.h*

```
//these are used for normal tracing operations.  description is a printf format string
//they default the trace level to IT_TRACE, if you want to use another level see the previous set
#define TRACELOG(description) \
    IT_LOG_GUARDED(0, IT_TRACE, description)

#define TRACELOG0(description) \
    IT_LOG_GUARDED(0, IT_TRACE, description)

#define TRACELOG1(description, Arg1) \
    IT_LOG_GUARDED1(0, IT_TRACE, description, Arg1)

#define TRACELOG2(description, Arg1, Arg2) \
    IT_LOG_GUARDED2(0, IT_TRACE, description, Arg1, Arg2)

#define TRACELOG3(description, Arg1, Arg2, Arg3) \
    IT_LOG_GUARDED3(0, IT_TRACE, description, Arg1, Arg2, Arg3)

#define TRACELOG4(description, Arg1, Arg2, Arg3, Arg4) \
    IT_LOG_GUARDED4(0, IT_TRACE, description, Arg1, Arg2, Arg3, Arg4)

#define TRACELOG5(description, Arg1, Arg2, Arg3, Arg4, Arg5) \
    IT_LOG_GUARDED5(0, IT_TRACE, description, Arg1, Arg2, Arg3, Arg4, Arg5)

#endif
```

IT_Logging Overview

The `IT_Logging` module is the centralized point for controlling all logging methods. The `LogStream` interface controls how and where events are received.

The `IT_Logging` module also uses the following common data types, static method, and macros.

Table 12: *IT_Logging Common Data Types, Methods, and Macros*

Common Data Types	Methods and Macros
`ApplicationId`	`format_message()`
`EventId`	
`EventParameters`	`IT_LOG_MESSAGE()`
`EventPriority`	`IT_LOG_MESSAGE_1()`
`SubsystemId`	`IT_LOG_MESSAGE_2()`
`Timestamp`	`IT_LOG_MESSAGE_3()`
	`IT_LOG_MESSAGE_4()`
	`IT_LOG_MESSAGE_5()`

IT_Logging::ApplicationId Data Type

```
//IDL
typedef string ApplicationId;
```

An identifying string representing the application that logged the event.

For example, a Unix and Windows `ApplicationId` contains the host name and process ID (PID) of the reporting process. Because this value can differ from platform to platform, streams should only use it as informational text, and should not attempt to interpret it.

IT_Logging::EventId Data Type

```
//IDL
typedef unsigned long EventId;
```

An identifier for the particular event.

IT_Logging::EventParameters Data Type

```
//IDL
typedef CORBA::AnySeq EventParameters;
```

A sequence of locale-independent parameters encoded as a sequence of `Any` values.

IT_Logging::EventPriority Data Type

```
//IDL
typedef unsigned short EventPriority;
```

Specifies the priority of a logged event. These can be divided into the following categories of priority.

Information	A significant non-error event has occurred. Examples include server startup/shutdown, object creation/deletion, and information about administrative actions. Informational messages provide a history of events that can be invaluable in diagnosing problems.
Warning	The subsystem has encountered an anomalous condition, but can ignore it and continue functioning. Examples include encountering an invalid parameter, but ignoring it in favor of a default value.
Error	An error has occurred. The subsystem will attempt to recover, but may abandon the task at hand. Examples include finding a resource (such as memory) temporarily unavailable, or being unable to process a particular request due to errors in the request.
Fatal Error	An unrecoverable error has occurred. The subsystem or process will terminate.

The possible values for an `EventPriority` consist of the following:

```
LOG_NO_EVENTS
LOG_ALL_EVENTS
LOG_INFO_LOW
LOG_INFO_MED
LOG_INFO_HIGH
LOG_INFO  (LOG_INFO_LOW)
```

```
LOG_ALL_INFO
LOG_WARNING
LOG_ERROR
LOG_FATAL_ERROR
```

A single value is used for `EventLog` operations that report events or `LogStream` operations that receive events. In filtering operations such as `set_filter()`, these values can be combined as a filter mask to control which events are logged at runtime.

IT_Logging::format_message()

```
// C++
static char* format_message(
    const char* description,
    const IT_Logging::EventParameters& params
);
```

Returns a formatted message based on a format description and a sequence of parameters.

Parameters

Messages are reported in two pieces for internationalization:

`description`	A locale-dependent string that describes of how to use the sequence of parameters in `params`.
`params`	A sequence of locale-dependent parameters.

`format_message()` copies the `description` into an output string, interprets each event parameter, and inserts the event parameters into the output string where appropriate. Event parameters that are primitive and `SystemException` parameters are converted to strings before insertion. For all other types, question marks (`?`) are inserted.

IT_Logging::SubsystemId Data Type

```
//IDL
typedef string SubsystemId;
```

An identifying string representing the subsystem from which the event originated. The constant `_DEFAULT` may be used to enable all subsystems.

IT_Logging::Timestamp Data Type

```
//IDL
typedef unsigned long  Timestamp;
```

The time of the logged event in seconds since January 1, 1970.

IT_Logging::LogStream Interface

Each of the Artix logging plug-ins implements the IT_Logging::LogStream interface. The LogStream interface allows an application to intercept events and write them to some concrete location via a stream.

IT_Logging::EventLog objects maintain a list of LogStream objects. You register a LogStream object from an EventLog using register_stream().

The complete LogStream interface is as follows:

```
// IDL in module IT_Logging
interface LogStream {
    void report_event(
        in ApplicationId   application,
        in SubsystemId     subsystem,
        in EventId         event,
        in EventPriority   priority,
        in Timestamp       event_time,
        in any             event_data
    );

    void report_message(
        in ApplicationId   application,
        in SubsystemId     subsystem,
        in EventId         event,
        in EventPriority   priority,
        in Timestamp       event_time,
        in string          description,
        in EventParameters parameters
    );
};
```

These operations are described in detail as follows:

LogStream::report_event()

```
// IDL
void report_event(
    in ApplicationId   application,
    in SubsystemId     subsystem,
    in EventId         event,
    in EventPriority   priority,
    in Timestamp       event_time,
    in any             event_data
);
```

Reports an event and its event-specific data to the log stream.

Parameters

`application`	An ID representing the reporting application.
`subsystem`	The name of the subsystem reporting the event.
`event`	A unique ID defining the event.
`priority`	The event priority.
`event_time`	The time when the event occurred.
`event_data`	Event-specific data.

See also `IT_Logging::EventLog::report_event()`
`IT_Logging::LogStream::report_message()`

LogStream::report_message()

```
// IDL
void report_message(
    in ApplicationId   application,
    in SubsystemId     subsystem,
    in EventId         event,
    in EventPriority   priority,
    in Timestamp       event_time,
    in string          description,
    in EventParameters parameters
);
```

Reports an event and message to the log stream.

Parameters

`application`	An ID representing the reporting application.
`subsystem`	The name of the subsystem reporting the event.
`event`	The unique ID defining the event.
`priority`	The event priority.
`event_time`	The time when the event occurred.
`description`	A string describing the format of `parameters`.
`parameters`	A sequence of parameters for the log.

See also `IT_Logging::EventLog::report_message()`

```
IT_Logging::LogStream::report_event()
```

Example

Controlling Application Logging

This example shows application logging enable by including the `xmlfile_log_stream` plugin in the `orb_plugins` list (this plugin is included in the default `orb_plugins` list, though it is not included in the `orb_plugins` lists within many of the demo program configuration scopes). If you want to enable logging to an XML file for the applications you develop, include this plugin in your orb_plugins list.

To enable usage of the xmlfile_log_stream plugin, several other configuration variables must also be set. These variable are all set within the default/global scope in the `artix.cfg` file:

```
plugins:xmlfile_log_stream:shlib_name =
  “it_xmlfile”;

plugins:xmlfile_log_stream:filename =
  “artix_logfile.xml”;
# default: it_bus.log

plugins:xmlfile_log_stream:max_file_size =
  “2000000”;
# default: 2 mb

plugins:xmlfile_log_stream:use_pid =
  “false”;
# default: false

# standard logging setting; logs errors and warnings
event_log:filters =
    [“*=FATAL+ERROR+WARNING”];

# very detailed logging
#event_log:filters = [“*=*”];

# transport buffer logging
#event_log:filters =
    [“*=FATAL+ERROR+WARNING+INFO_MED”];

# high level informational logging
#event_log:filters =
    [“*=FATAL+ERROR+WARNING+INFO_HI”];
```

Using the Logging Functionality

The default configuration settings enable logging of only serious errors and warnings. If you want more exhaustive information, you should either select a different filter list at the default scope, or include a more expansive `event_log:filters` configuration variable within your configuration scope.

If you have trouble running any of the demos, you should enable a high level of logging, whichrequires adding the xmlfile_log_stream plugin to the orb_plugins list and selecting the desired reporting level.

Performance Logging

Overview

The performance logging plugins allow applications based on IONA products to integrate effectively with Enterprise Management Systems (EMSs) such as IBM Tivoli™, HP OpenView™, CA Unicenter™ or BMC Patrol™.

Performance logging lets you see how each server is responding to load. These plugins log this data to file or syslog. Your EMS can read the performance data from the logs and initiate appropriate actions. For example, issuing a restart to a server that has become unresponsive, or starting a new replica for an overloaded cluster.

Configuration

The performance logging component consist of three plugins:

- The response time logger plugin
- The request counter plugin
- The collector plugin

The response time logger plugin monitors response times of requests as they pass through ART binding chains. It can be used to collect response times for CORBA, RMI-IIOP or HTTP calls in IONA's CORBA and J2EE products. The request counter plugin performs the same function for Artix.

The collector plugin periodically harvests data from the response time logger and request counter plugins and logs the results. To monitor the performance of CORBA or J2EE requests (made in the context of IONA's Application Server Platform), you must perform the following steps to reconfigure the Application Server Platform:

Add `it_response_time_logger` to the orb_plugins list for the server you wish to instrument. Add `it_reponse_time_logger` to the server and servlet binding lists for that server. For example:

```
binding:servlet_binding_list= [
"it_response_time_logger + it_servlet_context + it_character_encoding
+ it_locale + it_naming_context + it_exception_mapping + it_http_sessions
+ it_web_security + it_servlet_filters + it_web_redirector + it_web_app_activator "
];
binding:server_binding_list=[
"it_response_time_logger+it_naming_context+CSI+j2eecsi+OTS+it_security_role_mapping",
"it_response_time_logger+it_naming_context+OTS+it_security_role_mapping",
"it_response_time_logger+it_naming_context + CSI+j2eecsi+it_security_role_mapping",
"it_response_time_logger+it_naming_context+it_security_role_mapping",
"it_response_time_logger+it_naming_context",
"it_response_time_logger"
];

orb_plugins=[
"it_servlet_binding_manager", "it_servlet_context",
"it_http_sessions", "it_servlet_filters", "http",
"it_servlet_dispatch", "it_exception_mapping", "it_naming_context",
"it_web_security", "it_web_app_activator",
"it_default_servlet_binding", "it_security_service", "it_character_encoding",
"it_locale", "it_classloader_servlet","it_classloader_mapping",
"it_web_redirector", "it_deployer",
"it_response_time_logger"
];
```

Monitoring an Artix Server

Configuring the collector plugin

You can configure the collector plugin to log data either to a file or to syslog. The following example results in performance data being logged to `/var/log/my_app/perf_logs/treasury_app.log` every 90 seconds (if you do not specify the period, it defaults to 60 seconds):

```
plugins:it_response_time_collector:period = "90";

plugins:it_response_time_collector:filename =
   "/var/log/my_app/perf_logs/treasury_app.log";
```

You can also configure the collector to log to a syslog daemon or Windows Event Log:

```
plugins:it_response_time_collector:system_logging_enabled = "true";
plugins:it_response_time_collector:syslog_appid = "treasury";
```

`syslog_appid` lets you specify the application name, which is prepended to all syslog messages. If you do not specify a `syslog_appid`, it defaults to "iona".

You can cause your EMS to monitor a cluster of servers by configuring multiple servers to log to the same file. If the servers are running on different hosts, then the log file's location must be on an NFS mounted or shared directory.

Alternatively, you can use `syslogd` as a mechanism for monitoring a cluster, by choosing one `syslogd` to act as the central logging server for the cluster. For example, to use the host `teddy` as the central log server, edit the `/etc/syslog.conf` file for each host that runs a server replica, and add:

```
# Substitute the name of your log server
user.info @teddy
```

Some syslog daemons do not accept log messages from other hosts by default. In this case it may be necessary to restart the `syslogd` on `teddy` with a special flag to allow remote log messages. Consult the man pages on your system to determine whether this is necessary and what flags to use.

Logging Formats

Performance data is logged in a well-defined format. For CORBA and J2EE applications based on IONA's Application Server Platform, this format is:

```
YYYY-MM-DDTHH:MM:SS [operation=name] count=n avg=n max=n min=n
```

- `operation` is the name of the operation for CORBA invocations or the URI for requests on servlets.
- `count` is the number of times this operation or URI was logged during the last interval.
- `avg` is average response time (in milliseconds) for this operation or URI during the last interval.
- `max` is the longest response time (in milliseconds) for this operation or URI during the last interval.

- `min` is the shortest response time (in milliseconds) for this operation or URI during the last interval.

The format for Artix log messages is:

```
YYYY-MM-DDTHH:MM:SS [namespace=nnn service=sss port=ppp operation=name] count=n avg=n max=n min=n
```

- `namespace` is an Artix namespace.
- `service` is an Artix service.
- `port` is an Artix port.
- `operation` is the name of the operation for CORBA invocations or the URI for requests on servlets.
- `count` is the number of times this operation or URI was logged during the last interval.
- `avg` is average response time (in milliseconds) for this operation or URI during the last interval.
- `max` is the longest response time (in milliseconds) for this operation or URI during the last interval.
- `min` is the shortest response time (in milliseconds) for this operation or URI during the last interval.

The combination of namespace, service and port denote a unique Artix Service Access Point.

Load Balancing

Artix solutions can be configured to balance workloads among a number of servers.

Overview

Artix provides two methods for balancing workloads among a number of servers. One uses the lightweight Artix locator to load balance among a groups of Artix enabled servers. The second leverages IONA's Application Server Platform's location services to load balance among Artix servers that use CORBA as their communications medium.

In this chapter

This chapter discusses the following topics:

Load Balancing with the Artix Locator

Overview

The Artix locator provides a lightweight mechanism for balancing workloads among a group of servers. When a number of servers with the same service name register with the Artix locator, it automatically creates a list of the references and hands out the references to clients using a round robin algorithm. This process is invisible to both the clients and the servers.

Advantages

Using the Artix locator to load balance provides several advantages over using CORBA load balancing. Chief among these is that using the Artix locator allows you to use all of the supported Artix transports. Also, the Artix locator does not require you to have another middleware platform installed into your environment.

Starting to load balance

To begin load balancing with the Artix locator you must deploy an Artix locator, configure your servers to load the locator plug-ins, and design your clients to look up their server references from the Artix locator. For information on doing this see "Using the Artix Locator Service" on page 93.

Once the locator is deployed and your servers are properly configured, you need to bring up a number of instances of the same service. This can be accomplished by one of two methods depending on your system topology:

1. Create an Artix contract with a number of ports for the same service and have each server instance startup on a different port.
2. Create a number of copies of the Artix contract defining the service, change the port information so each copy has a separate port address, and then bring up each server instance using a different copy of the Artix contract.

Note: The locator uses the service name specified in the <`service`> tag of the server's Artix contract to determine if it is part of a group. It is recommended that if you are using the Artix locator to load balance, your services should be associated with the same binding and logical interface.

As each server starts up it will automatically register with the locator. The locator will recognize that the servers all have the same service name specified in their Artix contracts and will create a list of references for these server instances.

As clients make requests for the service, the locator will cycle through the list of server instances to hand out references.

Load Balancing with CORBA

Overview

If an Artix SAP is mapped to a CORBA service, and that CORBA service is accessible via IONA's Application Server Platform 6.0 Service Pack 1 (or later), the implementation of that service can be load balanced using the Application Server Platform's locator service. In order to accomplish this, the Artix configuration file must duplicate some of the information from the Application Server Platform configuration domain, as described in the following steps.

For information on the load balancing feature of the Application Server Platform's load balancing features read the *Application Server Platform Administrator's Guide*.

Configuration Steps

The following steps work with an Application Server Platform installation that uses either file-based configuration or a configuration repository. However, because Artix supports only file-based configuration, the relevant configuration information must be inserted into the `artix.cfg` file. The following configuration example assumes that an Application Server Platform domain exists, and that the locator service is run from this domain:

1. From the `domain.cfg` file, obtain the following configuration information and add it to `artix.cfg` file.

```
initial_references:IT_NodeDaemon:reference =
   "IOR:000000000000002149444c3a49545f4e6f64654461656d6f6e2f4e6f64654461656d6f6e3a312e3000000000
   00000001000000000000007600010200000000008686f726174696f00782800000000001d3a3e0233310c6e6f64655
   f6461656d6f6e000a4e6f64654461656d6f6e000000000000030000000100000018000000000001000100000000000
   0101040000000100010109000001a000000040100000000000000600000006000000000011";
```

2. Create an `ORBname` for each Artix SAP that participates in load balancing. For example:

```
itadmin orbname create demos.clustering.server_1
itadmin orbname create demos.clustering.server_2
itadmin orbname create demos.clustering.server_3
```

3. Create a POA that declares these `ORBnames` as replicas, and specify either round-robin or random load balancing. For example:

```
itadmin poa create -replicas
   demos.clustering.server_1,demos.clustering.server_2,demos.clustering.server_3
   -load_balancer round_robin ClusterDemo
```

The POA name (`ClusterDemo`) is expressed in WSDL as:

```
<corba:policy persistent="true" serviceid="service_id" poaname="ClusterDemo"/>
```

You can choose any POA name; however, the POA name you register using `itadmin` must be the same name you declare in the WSDL file.

When `corba:policy persistent=true` is specified, you must also specify `serviceid`. Failure to specify `serviceid` will either result in an IOR that cannot be used for load balancing, or a process that outlives the POA.

To run such a ClusterDemo, you start the CORBA servers that underlie the Artix SAP as follows:

```
Server -ORBname demos.clustering.server_1
Server -ORBname demos.clustering.server_2
Server -ORBname demos.clustering.server_3
```

When you run a client to connect to the Artix SAP, the first request goes to the first server (because `round_robin` load balancing was declared). If a second client is started, its request goes to the second server, and a third client’s request goes to the third server.

Replicated Application Server Platform services

If your Application Server Platform services are replicated, and if Artix is deployed on each of the machines on which those services are replicated, then the Artix SAPs themselves can be replicated and load-balanced. For example,

1. On the “master” machine (e.g., the machine that hosts the configuration repository), create an `ORBname` for each Artix SAP that participates in load balancing. For example:

```
itadmin orbname create demos.clustering.server_1
itadmin orbname create demos.clustering.server_2
itadmin orbname create demos.clustering.server_3
```

2. Create a POA that declares these `ORBnames` as replicas, and specify either round-robin or random load balancing. For example:

```
itadmin poa create -replicas
   demos.clustering.server_1,demos.clustering.server_2,demos.clustering.server_3
   -load_balancer round_robin ClusterDemo
```

3. On each machine that replicates the service, obtain the Node Daemon's initial reference and add it to the `artix.cfg` file on that machine.
4. Start a server on each machine, passing one of the three specified `ORBnames` to it (`clustering.server_1`, `demos.clustering.server_2`, `or demos.clustering.server_3`).

This service is now load balanced among the three replicated Artix SAPs. If one or two of these SAPs is killed, the client invocation is directed to the remaining machine(s).

Creating the load-balanced environment dynamically

It is possible to create a load balance environment without creating the POA or manually registering ORB names. To accomplish this:

1. On the master machine, obtain the Node Daemon initial reference and put it in the `artix.cfg` file.
2. Start the CORBA service, passing the same ORB name as that specified in the Artix client's WSDL contract. This ORB name is received by the Node Daemon, which creates a POA with that name. If you do not specify an ORB name, the name `WSORB` is used.
3. On the master machine, issue the following command in the Application Server Platform environment with the name you chose:

```
itadmin poa modify -allowdynreplicas yes POA_Name
```

4. On each of the slave machines where the service is replicated, obtain the Node Daemon initial reference from the Application Server Platform domain configuration and put it in the `artix.cfg` file.
5. On each of the slave machines where the service is replicated, start the CORBA service, using a *different* `ORBname` each time.

6. On the master machine, issue the following command in the Application Server Platform environment (inserting the type of load balancing and the ORBnames you have chosen):

```
itadmin poa modify -l <round_robin | random> POA_name
```

7. Start the Artix SAP.

Other load balancing features

In addition to POA name, the Application Server Platform configuration file can also affect load balancing by specifying:

1. Persistent or Transient POA policy
2. Object ID

These load-balancing-related configuration values can be specified in an Artix WSDL contract using WSDL extensions for CORBA ports:

The POA name can be specified as follows:

```
<corba:policy poaname="my_poa_name"/>
```

The default POA name is WSORB.

The POA persistence policy can be set as follows:

```
 <corba:policy persistent="true | false"/>
```

If this value is set to true, the POA policy is persistent. The default persistence value is false.

The Service ID can be set as follows:

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

Object ID is provided by the POA if the POA Policy SYSTEM_ID is set. Setting this to any string sets the POA policy USER_ID and uses the value provided as the object_id. If this is not set, the POA policy is SYSTEM_ID.

The following WSDL examples illustrate these points.

The contract fragment in Example 53 results in the following POA policy settings:

- PERSISTENT
- USER_ID
- POAName="master1"

- ObjectID="master1"

Example 53: *Setting the PERSISTENT POA policy*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
    <corba:policy persistent="true" poaname="master1" serviceID="master1"/>
  </port>
</service>
```

The contract fragment in Example 54 results in the following POA policy settings:

- TRANSIENT (Default)
- SYSTEM_ID (Default)
- POAName="master1"

Example 54: *Setting the POAName POA policy*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
    <corba:policy poaname="master1"/>
  </port>
</service>
```

The contract fragment in Example 55 results in a POA with the following policy settings:

- TRANSIENT (Default)
- USER_ID
- POAName="WSORB" (Default)
- ObjectID="master1"

Example 55: *Setting the USER_ID POA policy*

```
     <service name="BaseService">
         <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
            <corba:address location="file://master.ref"/>
            <corba:policy poaname="master1" serviceID="master1"/>
         </port>
     </service>
```

The contract fragment in Example 56 results in a POA with all default policies.

Example 56: *Default POA policies*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
  </port>
</service>
```

CHAPTER 10

Using the CORBA Plug-in

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:

- object references
- 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 13 lists the mappings for the supported IDL primitive types.

Table 13: *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`
`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`
`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 following CORBA types:

- `wchar`
- `wstring`
- `long double`

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 57.

Example 57: *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 58.

Example 58: *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 59.

Example 59: *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 14 shows the mappings from complex IDL types to XMLSchema, Artix CORBA type, and Artix C++ types.

Table 14: *Complex Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
`struct`	See Example 60	`corba:struct`	`IT_Bus::SequenceComplexType`
`enum`	See Example 61	`corba:enum`	`IT_Bus::AnySimpleType`
`fixed`	`xsd:decimal`	`corba:fixed`	`IT_Bus::Decimal`
`union`	See Example 66	`corba:union`	`IT_Bus::ChoiceComplexType`
`typedef`	See Example 69		
`array`	See Example 71	`corba:array`	`IT_Bus::ArrayT<>`
`sequence`	See Example 77	`corba:sequence`	`IT_Bus::ArrayT<>`
`exception`	See Example 80	`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 with 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 60:

Example 60: *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 61:

Example 61: *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 62, `myFixed`, would be

Example 62: *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 63.

Example 63: *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 64. 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 64: *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 65:

Example 65: *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 66:

Example 66: *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:unionbrach>` 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 65 on page 179, would be described with a CORBA type map entry similar to that shown in Example 67.

Example 67: *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 67: *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 68, can be described as

Example 68: *myLong IDL*

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

shown in Example 69:

Example 69: *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 mxOccurs attributes set to the value of the array's size. For example, consider an array, myArray, as defined in Example 70.

Example 70: *myArray IDL*

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

Its logical description will be similar to that shown in Example 71:

Example 71: *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 72:

Example 72: *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 73: *Multidimensional Array*

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

generates the following logical description:

Example 74: *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 75: *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 76, `longSeq` and `charSeq`:

Example 76: *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 77:

Example 77: *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 77 has a CORBA type map description similar to that shown in Example 78:

Example 78: *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 79, `idNotFound`,

Example 79: *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 80:

Example 80: *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 81:

Example 81: *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>
```

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 82 results in the CORBA typemap description shown in Example 83.

Example 82: *joeBinary schema description*

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

The resulting IDL for `joeBinary` is shown in Example 84.

Example 83: *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 84: *joeBinary IDL*

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

Attribute mapping

XMLSchema attributes are treated as normal elements in a CORBA structure. For example, the complex type, `madAttr`, described in Example 85 contains two attributes, `material` and `size`.

Example 85: *madAttr XMLSchema*

```
<complexType name="madAttr">
  <sequence>
    <element name="style" type="xsd:string" />
    <element name="gender" type="xsd:byte" />
  </sequence>
  <attribute name="material" type="xsd:string" />
  <attribute name="size" type="xsd:int" />
<complexType>
```

`madAttr` would generate the CORBA typemap shown in Example 86. Notice that `material` and `size` are simply incorporated into the madAttr structure in the CORBA typemap.

Example 86: *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="material" idltype="corba:string"/>
  <corba:member name="size" idltype="corba:long"/>
</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 87.

Example 87: *madAttr IDL*

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

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 88, contains a nested `choice`.

Example 88: *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 89, 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 89: *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 90.

Example 90: *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 `<extension>` tag. For example the complex type `seaKayak`, described in Example 91, inherits a number of fields from the complex type `kayak`.

Example 91: *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 inheritance described with the `<extension>` tag are mapped into CORBA, Artix flattens the inheritance. As shown in Example 92, Artix maps the inherited fields as normal members of the structure in the CORBA type map. The inheritance chain is not maintained.

Example 92: *seaKayak CORBA type map*

```
<corba:struct name="seaKayak" repositoryID="IDL:seaKayak:1.0" type="typens:seaKayak">
  <corba:element name="length" idltype="corba:long" />
  <corba:element name="width" idltype="corba:long" />
  <corba:element name="material" idltype="corba:string" />
  <cobra:element name="chines" idltype="corba:string" />
  <corba:element name="cockpitStyle" idltype="corba:string" />
</corba:struct>
<corba:struct name="kayak" repositoryID="IDL:seaKayak:1.0" type="typens:seaKayak">
  <corba:element name="length" idltype="corba:long" />
  <corba:element name="width" idltype="corba:long" />
  <corba:element name="material" idltype="corba:string" />
</corba:struct>
```

The IDL generated by Artix to handle complex schema types that use inheritance also flattens the inheritance as shown in Example 93.

Example 93: *seaKayak IDL*

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

Because the CORBA mappings break the inheritance chain, you must be careful about how data is exchanged between components using contracts with this type of mapping. While the service for which the original schema types were developed may treat certain objects as equivalent due to inheritance, the CORBA services using the contract do not and will not handle receiving the wrong data gracefully.

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 `nillabe="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 94.

Example 94: *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 95 shows the CORBA typemap for `webData`.

Example 95: *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 96 shows the IDL for `webData`.

Example 96: *webData IDL*

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

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.

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.

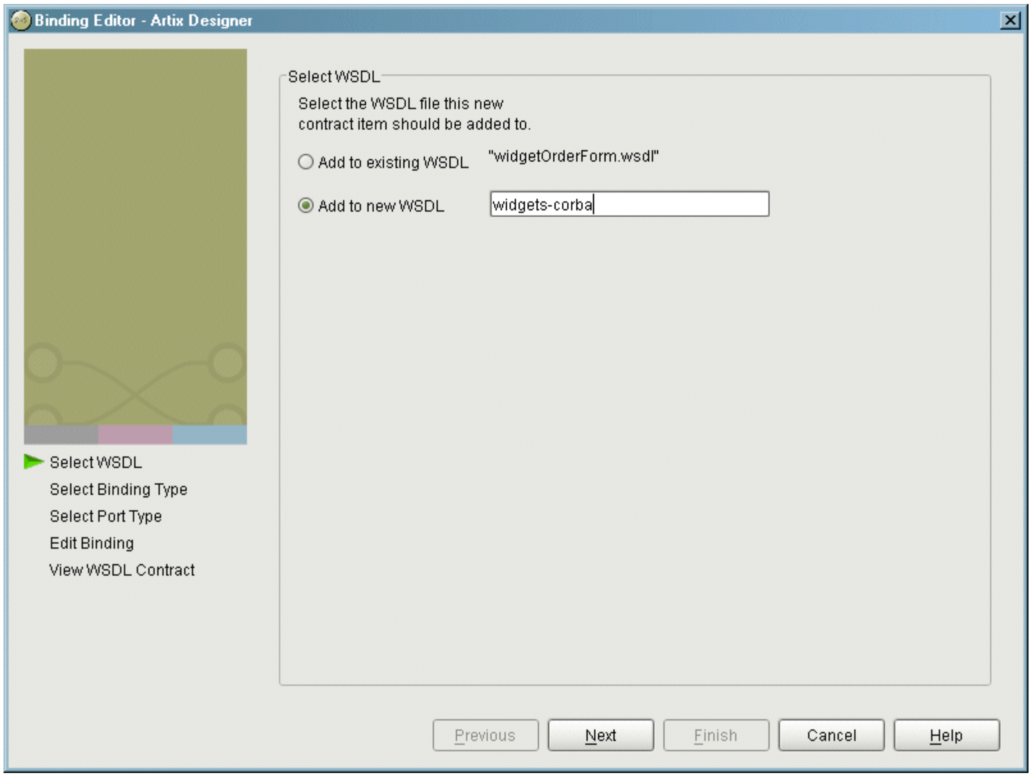
Using Artix Designer

The Binding Editor walks you through the generation of a CORBA binding based on your existing contract. It then generates a new contract containing the CORBA binding and the associated CORBA type map.

To add a CORBA binding to an Artix contract complete the following steps:

1. From the project tree, select the service to which you want to add the CORBA binding.
2. Select **Bindings|New Binding** from the **Contract** menu of the designer.

3. You will see a screen like Figure 12.



Select the WSDL file this new contract item should be added to.

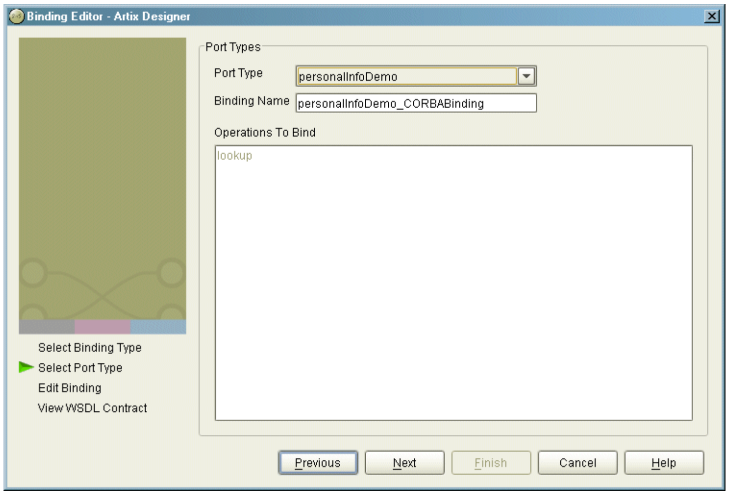
( ) Add to existing WSDL "widgetOrderForm.wsdl"

(●) Add to new WSDL widgets-corba

Figure 12: *Select WSDL location*

4. Select where to create the WSDL entry for the new binding.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Select **CORBA** from the list of possible bindings.

7. Click **Next** to select the interface you want mapped to the CORBA binding.
8. You will see a dialog similar to Figure 13.



Port Types

Port Type: personalInfoDemo

Binding Name: personalInfoDemo_CORBABinding

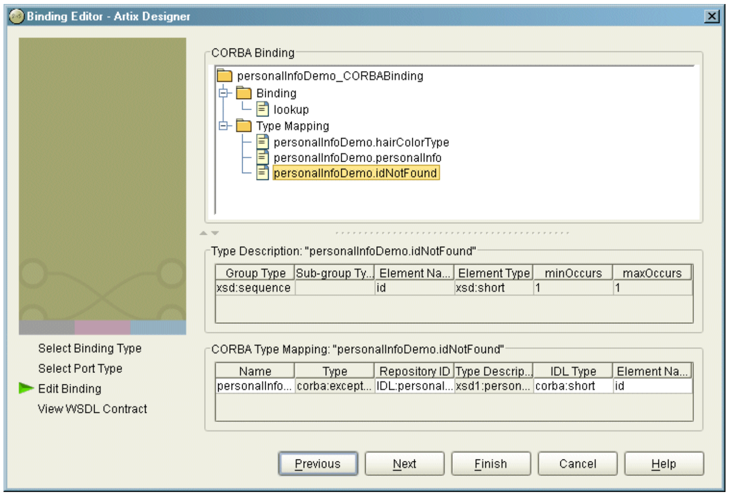
Operations To Bind

lookup

Figure 13: *Select Interface to Map to CORBA*

9. From the drop down list select the interface you want to map to the CORBA binding.
10. Enter the name for the new binding.
11. If there is more than one operation described in the interface, select the operation that are to be mapped into the CORBA binding.
12. Click **Next** to edit the new CORBA binding.

13. You will see a dialog similar to Figure 14.



CORBA Binding

```
personalInfoDemo_CORBABinding
├ Binding
│ └ lookup
└ Type Mapping
  ├ personalInfoDemo.hairColorType
  ├ personalInfoDemo.personalInfo
  └ personalInfoDemo.idNotFound
```

Type Description: "personalInfoDemo.idNotFound"

Group Type	Sub-group Ty...	Element Na...	Element Type	minOccurs	maxOccurs
xsd:sequence		id	xsd:short	1	1

CORBA Type Mapping: "personalInfoDemo.idNotFound"

Name	Type	Repository ID	Type Descrip...	IDL Type	Element Na...
personalInfo...	corba:except...	IDL:personal...	xsd1:person...	corba:short	id

Figure 14: *Edit the CORBA Binding*

14. Examine the different elements of the binding by selecting them from the tree at the top of the dialog.
15. Edit the values shown in white if they are not correct.
16. When you are finished editing the binding, click **Next**.
17. Review the newly created contract containing the new CORBA binding.
18. If the contract is correct, click **Finish**.

When you have completed creating the new CORBA binding the contract describing the binding and the CORBA type map is added to the project tree under the selected service. This new contract will not contain a CORBA port description. For details on adding a CORBA port description see "Adding a CORBA Port" on page 204.

Using the command line

The wsdltocorba tool also 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]
   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.

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 97.

Example 97: *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 "Using the CORBA Naming Service" on page 220.

- 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" />
```

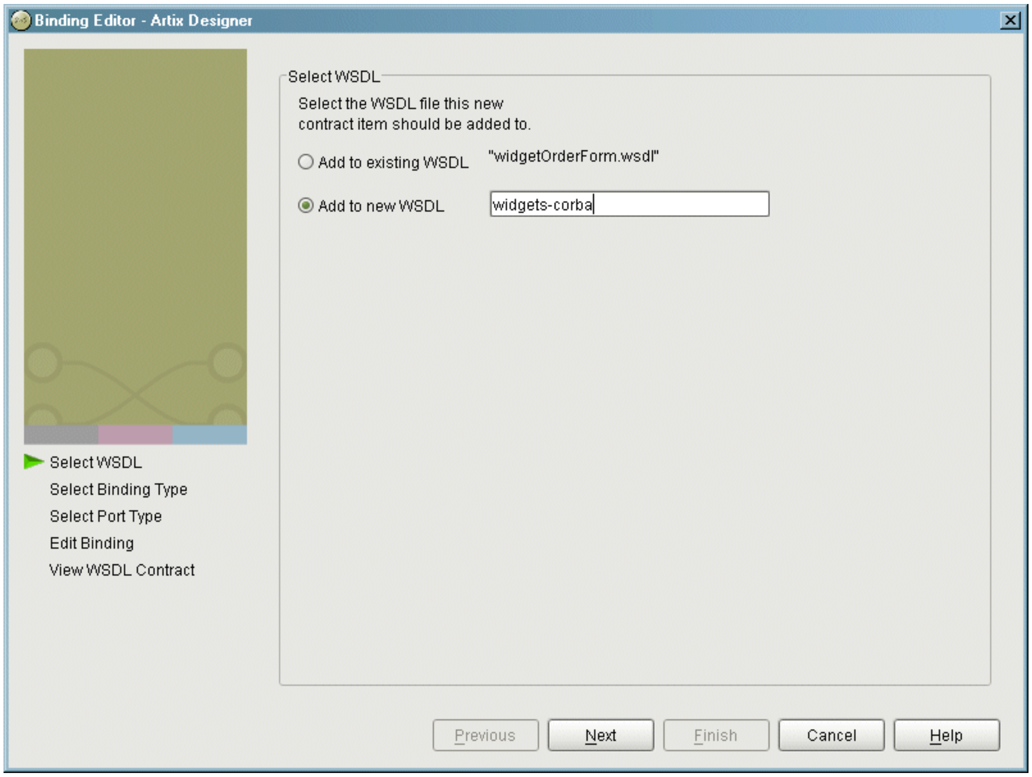
This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Procedure

To add a CORBA port to your service contract using the GUI, complete the following steps:

1. From the project tree, select the contract to which you want to add the CORBA port.
2. Select **Services|New Service** from the **Contract** menu of the designer.

3. You will see a screen like Figure 12.



Select WSDL

Select the WSDL file this new contract item should be added to.

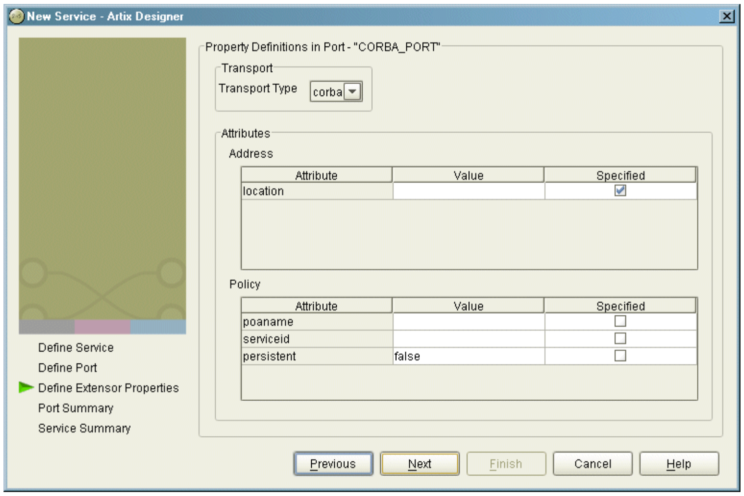
( ) Add to existing WSDL "widgetOrderForm.wsdl"

[✓ Add to new WSDL] widgets-corba

Figure 15: *Select WSDL Location*

4. Select where to create the WSDL entry for the new service.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Enter a unique name for the new service.

7. Click **Next**.
8. Enter a name for the new CORBA port that is being created.
9. From the drop down list, select the binding that the port is going to expose.
10. Click **Next**.
11. You will see a dialog similar to Figure 16.



New Service - Artix Designer

Property Definitions in Port - "CORBA_PORT"

Transport

Transport Type: corba

Attributes

Address

Attribute	Value	Specified
location		✓

Policy

Attribute	Value	Specified
poaname		☐
serviceid		☐
persistent	false	☐

Figure 16: *Edit CORBA Port Properties*

12. From the drop down list in the **Transport** box, select **corba**.
13. In the **Address** table, enter the CORBA address in the line for **Location**.
14. If you want to set any of the supported POA policies, place a check in the **Specified** box on the appropriate line in the **Policy** table and enter a valid value.
15. Click **Next**.
16. Review the settings for the new CORBA port.
17. If it is correct, click **Next**.
18. Review the settings for the new service in which the CORBA port is described.

19. If it is correct, click **Finish**.

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to Example 98:

Example 98: *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.

Using Artix Designer

To generate IDL from the Artix Designer complete the following steps:

1. Select the Development icon under the service for which you are going to generate IDL.

   Note: The service must have a CORBA binding defined in one of its associated contracts to generate IDL.

2. From the drop down list next to **Development Environment** select **IDL**.
3. Enter the name and location of the file to which the generated IDL will be generated.
4. Click **OK**.

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.id`l:

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

  S get_op();
};
```

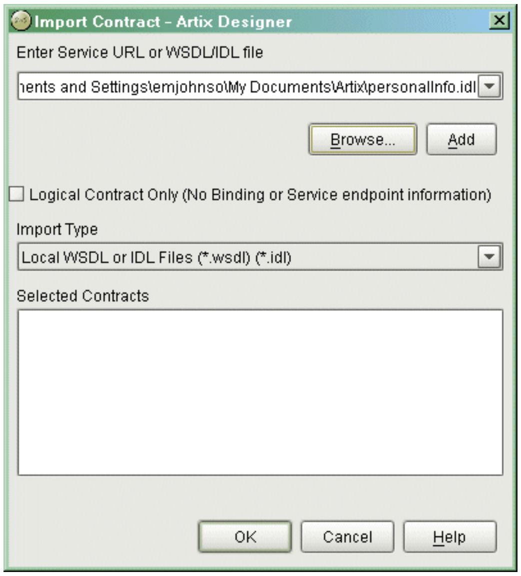
The IDL compiler does not generate any corresponding contract information for the structure `S` because it contains a member that uses an object reference. Similarly, the IDL complier does not generate any contract information for the operation `get_op()` because it references structure `S`.

Using Artix Designer

The Artix Designer imports IDL files, generates a new Artix contract to describe the CORBA service represented by the IDL, and adds the new contract to the project tree.

To import an IDL file into Artix Designer complete the following steps:

1. Select either a contracts folder or a service node from the project tree.
2. Select **Import...** from the **Contract** menu.
3. You will see a dialog similar to Figure 17.


Import Contract - Artix Designer
Enter Service URL or WSDL/IDL file
Browse...
Add
Import Type
Selected Contracts
OK
Cancel
Help

hents and Settings\emjohnso\My Documents\Artix\personalInfo.idl

Logical Contract Only (No Binding or Service endpoint information)

Local WSDL or IDL Files (*.wsdl) (*.idl)

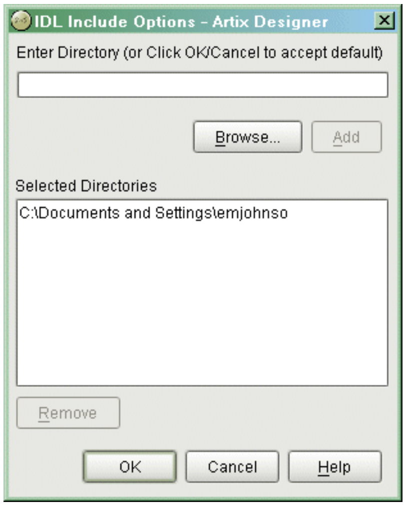
Figure 17: *IDL Import*

4. Click **Browse** to locate the IDL file.
5. Select **Local WSDL or IDL FIles** from the **Import Type** drop down list.
6. If you only wish to generate the logical portion of the contract select **Logical Contract Only**.

   Note: If this option is selected the generated contracts will not contain any binding, CORBA typemap, or transport information.

7. Click **Add** to move the IDL file into the **Selected Contracts** list.

8. A dialog similar to Figure 18 will appear.



Enter Directory (or Click OK/Cancel to accept default)

Selected Directories

C:\Documents and Settings\emjohnso

Figure 18: *IDL Include Directories*

9. Enter the names of the directories to search for included IDL files.
10. Click **Add** to add a directory to the list.
11. When finished adding directories, click **OK**.
12. Repeat steps 4 through 11 until you have added all of the IDL files to import.
13. Click **OK**.

One contract will be added to the project tree under the selected folder or service for each IDL file imported. The contracts will include a CORBA binding, a CORBA type map, and a CORBA port description. You will need to add location information to the CORBA port before you can deploy a service using the CORBA port. For information on adding a location to the CORBA port see "Address specification" on page 204.

From 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] idlfile
```

The command has the following options:

`-wsdl`	Specifies that WSDL is to be generated. This flag is required.
`-aaddress`	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
`-ffile`	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.
`-Odir`	Specifies the directory into which the WSDL file is written.
`-turi`	Specifies the URI for the `corbatm` namespace. This overrides the default.
`-stype`	Specifies the XMLSchema type used to map the IDL `sequence<octet>` type. Valid values are `base64Binary` and `hexBinary`. The default is `base64Binary`.
`-rfile`	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`.
`-Lfile`	Specifies that the logical portion of the generated WSDL specification into is written to *file*. *file* is then imported into the default generated file.
`-Pfile`	Specifies that the physical portion of the generated WSDL specification into is written to *file*. *file* is then imported into the default generated file.

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 99.

Example 99: *personalInfoService Interface*

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

  struct personalInfo
  {
    string name;
    long age;
    hairColorType hairColor;
  };

  exception idNotFound
  {
    short id;
  };

  personalInfo lookup(in long empId)
  raises (idNotFound);
};
```

To generate the contract, you run it through the IDL compiler using either the GUI or the command line. The resulting contract is similar to that shown in Example 100.

Example 100:*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>
      <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>
```

Example 100:*personalInfoService Contract*

```
<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>
    <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>
```

Example 100:*personalInfoService Contract*

```
    <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 101.

Example 101:*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*.

Using the CORBA Naming Service

Overview

In order to fully integrate with deployed CORBA systems, Artix can use a CORBA naming service that supports the `CosNaming` interface. Doing so requires editing the port information in the service's contract and modifying the Artix configuration.

Servers

To specify that an Artix instance (acting as proxy for a server) is to use the COBRA naming service, you edit the `<corba:address>` element of the CORBA port. In place of the file name used in the `location` attribute, specify a `corbaname`. For example, to specify that the converter server publishes its IOR to the CORBA naming service, specify the `<corba:address>` as follows:

```
<corba:address location="corbaname:rir:/NameService#personalInfoService"/>
```

This registers the server in the name service under the name `personalInfoService`.

Clients

An Artix instance (acting as a proxy for a client) can also use the `<corba:address>` element to specify what name to look up in the CORBA name service. The name the client looks up in the name service is the string after the `#` in the specified location. For example, a client using the `<corba:address>` shown above in "Servers" looks up the IOR for an object named `personalInfoService`.

Configuration

Artix applications that wish to use a CORBA name service must be configured to load a name resolver plug-in and have an initial reference for the running name service.

To modify the Artix configuration do the following:

1. Open the Artix configuration file, `IT_PRODUCT_DIR\artix\1.2\etc\artix.cfg`, in a text editor.
2. In the global scope, add the following lines:

```
initial_references:NameService:reference="corbaloc::localhost:portNumber/NameService";
url_resolvers:corbaname:plugin="naming_resolver";
plugins:naming_resolver:shlib_name="it_naming";
```

   portNumber is the number of the port on which the name service is running.

For more information on configuring Artix, see “Configuration” on page 27.

Embedding Artix in a CORBA Application

Overview

Artix, because it is built on IONA's flexible ART platform, can be embedded within any CORBA application implemented using IONA's Applications Server Platform 6.0 or later without modifying any of the CORBA applications code. Embedding Artix is done by altering the applications configuration to load the required Artix plug-ins.

Embedding Artix into your CORBA application has several advantages:

- You do not need a separate process to route messages to the non-CORBA pieces of your application.
- You improve messaging performance over using the Artix standalone service.
- You can still code using a familiar paradigm and realize the benefits of using Artix.
- You can leverage all of the CORBA infrastructure to provide enterprise level qualities of service and management.

CORBA client applications

To embed Artix into a CORBA client application you need to do the following:

1. Create an Artix contract that fully describes the interfaces, bindings, transports, and routing rules used in your Artix application.
2. Edit the configuration scope for your CORBA client so that the ORB plug-ins list contains the required Artix plug-ins to support the bindings and transports used by your Artix application.

   For example, if your CORBA client will be interacting with a sever using SOAP over WebSphere MQ your ORB plug-in list would be similar to the one in Example 102 on page 223. Note that the required Artix plug-ins for the SOAP binding, the WebSphere MQ transport, CORBA, and routing are highlighted.
3. Make an entry for `plugins:routing:wsdl_url` that specifies where the Artix applications contract resides.

In Example 102, the Artix contract describing the application is stored in `/artix/wsdlRepos/scoreBox.wsdl`.

Example 102:*Embedded Artix orb_plugin list*

```
corba_client.artix
{
  orb_plugins=["iiop_profile", "giop", "soap", "mq", "ws_orb",
   "routing"];
  plugins:routing:wsdl_url="/artix/wsdlRepos/scoreBox.wsdl";
}
```

4. When you start your CORBA client ensure that you start it using the proper ORB name to load the Artix plug-ins.

   For a client that uses the configuration shown in Example 102, you would start the client with the following command:

   ```
   client -ORBname corba_client.artix
   ```

CORBA server applications

To embed Artix into a CORBA server that uses the routing plug-in there are two caveats:

- Your CORBA server must generate persistent object references.
- Your CORBA server must run one time to export the persistent references and then be restarted for the Artix routing plug-in to work.

The routing plug-in requires valid object references to properly load itself and when embedded into the CORBA server, the routing plug-in is loaded by the ORB before any object references are generated. By using persistent object references and pregenerating them before fully deploying the server, as when using the naming service, you satisfy the routing plug-in.

Complete the following steps to configure a CORBA server to embed Artix:

1. Create an Artix contract that fully describes the interfaces, bindings, transports, and routing rules used in your Artix application.
2. Edit the configuration scope for your CORBA server so that the ORB plug-ins list contains the required Artix plug-ins to support the bindings and transports used by your Artix application.

   For example, if your CORBA server will be interacting with a client using SOAP over WebSphere MQ your ORB plug-in list would be similar to the one in Example 103 on page 224. Note that the required

Artix plug-ins for the SOAP binding, the WebSphere MQ transport, CORBA, and routing are highlighted.

3. Make an entry for `plugins:routing:wsdl_url` that specifies where the Artix applications contract resides.

   In Example 103, the Artix contract describing the application is stored in `/artix/wsdlRepos/scoreBox.wsdl`.

4. Edit the server's client binding list, `binding:client_binding_list`, so that none of the listed bindings use `POA_Coloc`.

   The configuration scope in Example 103 shows a client binding list that does not use `POA_Coloc`. The default client binding list includes entries for `"OTS+POA_Coloc"` and `"POA_Coloc"`.

Example 103:*Embedded Artix Server Configuration*

```
corba_server.artix
{
  orb_plugins=["iiop_profile", "giop", "soap", "mq", "ws_orb",
   "routing"];
  plugins:routing:wsdl_url="/artix/wsdlRepos/scoreBox.wsdl";
  binding:client_binding_list=["OTS+GIOP+IIOP", "GIOP+IIOP"];
  binding:server_binding_list=["OTS"];
}
```

5. When you start your CORBA server ensure that you start it using the proper ORB name to load the Artix plug-ins.

   For a server that uses the configuration shown in Example 103, you would start the client with the following command:

   ```
   server -ORBname corba_server.artix
   ```

CHAPTER 11

Using the HTTP Plug-in

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 explains how to configure and extend a WSDL contract to use an HTTP port and 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:

- "What is HTTP?" on page 226.
- "Resources and URLs" on page 226.
- "HTTP transaction processing" on page 227.
- "Format of HTTP client requests" on page 227.
- "Format of HTTP server responses" on page 229.
- "HTTP properties" on page 230.

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 243 and "Server Transport Attributes" on page 266 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 243 and "Client Transport Attributes" on page 268 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 226, 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 226 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 227 for details of the format of a client request message. See "Format of HTTP server responses" on page 229 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 243 for details of the various SSL-related configuration attributes that can be used in extending a WSDL contract.

Adding an HTTP Port

Overview

You can configure an Artix WSDL contract with various extensions that support the use of an HTTP port with an Artix integration solution. When adding an HTTP port to a contract you can choose to specify whether or not HTTP connections should run securely (over SSL or TLS). This section describes how to use the **Artix Designer** GUI to add both secure and non-secure HTTP ports to WSDL contracts.

Note: This section is only relevant if you want to use HTTP with a payload format other than SOAP. If you are using SOAP over HTTP, see the "SOAP Payload Format" chapter of this guide.

In this section

This section discusses the following topics:

Adding an HTTP Port for Non-Secure Connections

Overview

This section describes how to use the **Artix Designer** GUI to add to a WSDL contract an HTTP port that does not enable secure connections. It discusses the following topics:

- "GUI steps" on page 234.
- "WSDL example" on page 237.

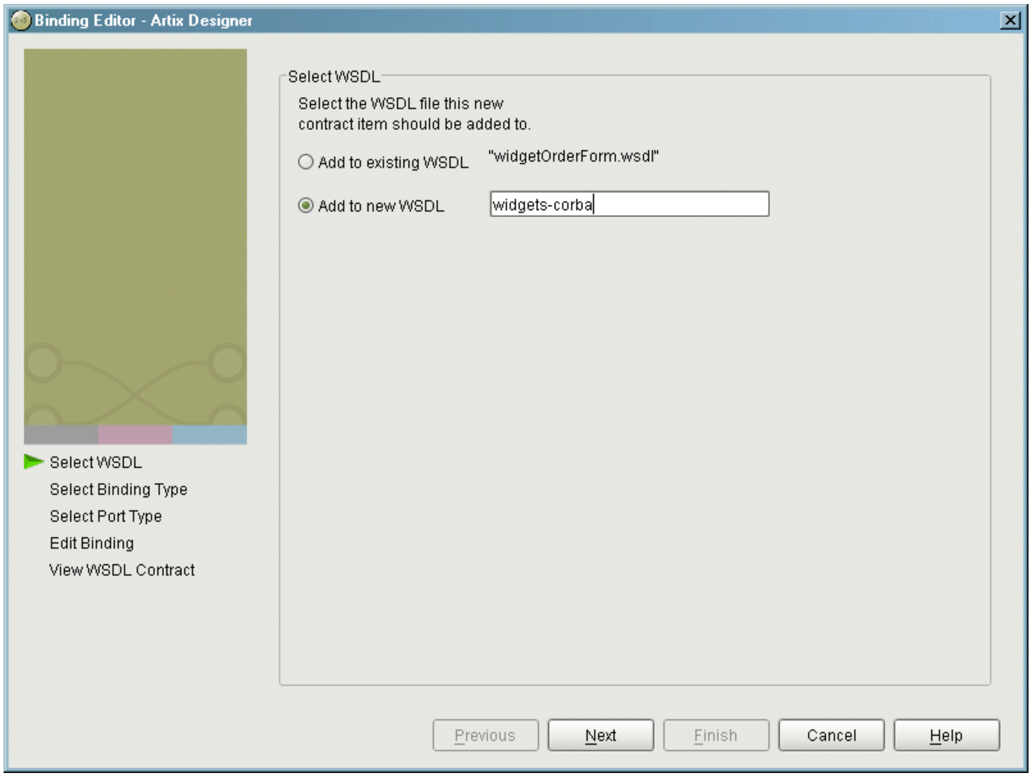
Note: This section deals specifically with how to set up port information within the `<service>` component of a WSDL contract. To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See the chapter relating to the payload format you are using for more details about setting up a binding for it in a WSDL contract.

GUI steps

To add an HTTP port to your service contract, using the **Artix Designer** GUI, complete the following steps:

1. From the project tree, select the contract to which you want to add the HTTP port.
2. Select **Services|New Service** from the **Contract** menu of the designer.

3. You will see a screen like Figure 19.



Select WSDL

Select the WSDL file this new contract item should be added to.

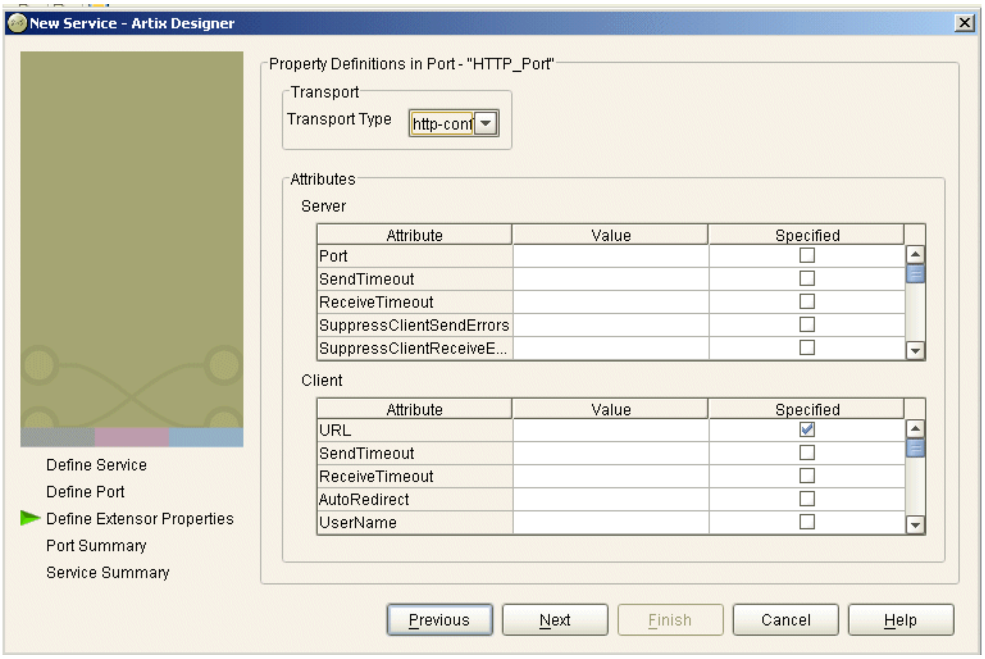
Add to existing WSDL "widgetOrderForm.wsdl"

[✓ Add to new WSDL] widgets-corba

Figure 19: *Select WSDL location*

4. Select where to create the WSDL entry for the new binding.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Enter a unique name for the new service.

7. Click **Next**.
8. Enter a name for the new HTTP port that is being created.
9. From the **Binding** drop down list, select the binding that the port is going to expose.
10. Click **Next**.
11. From the **Transport Type** drop down list, select **http-conf**. The screen then appears as shown in Figure 20.



Property Definitions in Port - "HTTP_Port"

Transport

Transport Type: http-conf

Attributes

Server

Attribute	Value	Specified
Port		☐
SendTimeout		☐
ReceiveTimeout		☐
SuppressClientSendErrors		☐
SuppressClientReceiveE...		☐

Client

Attribute	Value	Specified
URL		✓
SendTimeout		☐
ReceiveTimeout		☐
AutoRedirect		☐
UserName		☐

Figure 20: *Selecting an HTTP Transport Type*

Note: Except for the **URL** attribute in the **Client** configuration table, all attributes on this screen are optional.

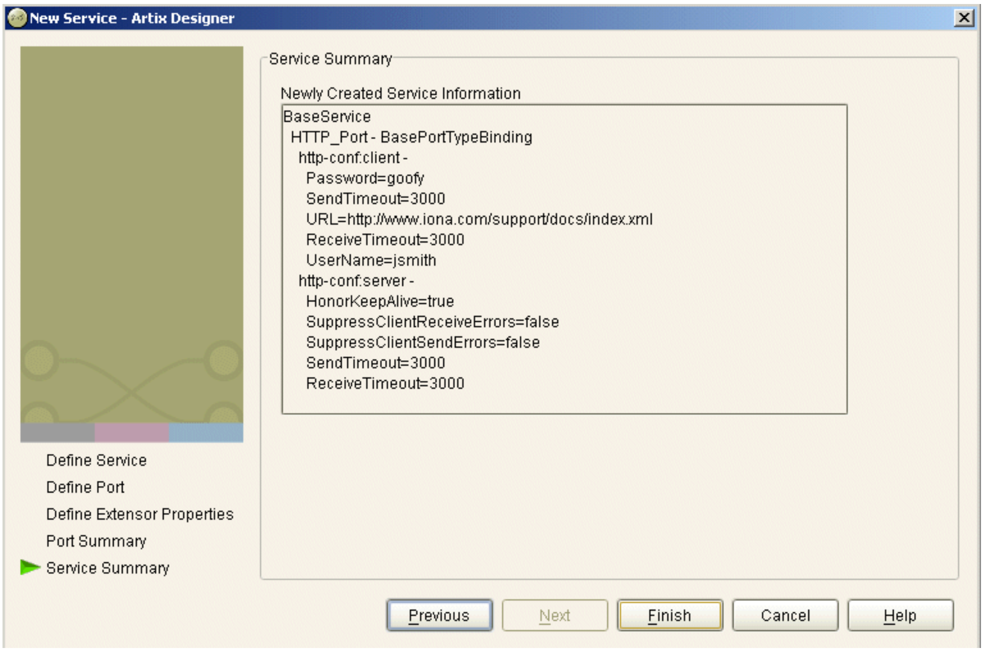
12. To specify a value for a particular attribute, place a check in the **Specified** box on the appropriate line, and type (or in the case of certain true or false attributes select) the value you want.

    Note: You must specify a value for the **URL** attribute. In this case, the URL you specify has a `http://` prefix. See "HTTP WSDL Extensions" on page 243 for details of all attributes.

13. Click **Next**.
14. Review the settings for the new HTTP port.
15. If it is correct, click **Next**.
16. Review the settings for the new service in which the HTTP port is described.
17. If it is correct, click **Finish**.

WSDL example

Figure 21 shows an example summary of HTTP configuration settings in the GUI.



Newly Created Service Information

```
BaseService
 HTTP_Port - BasePortTypeBinding
  http-conf:client -
   Password=goofy
   SendTimeout=3000
   URL=http://www.iona.com/support/docs/index.xml
   ReceiveTimeout=3000
   UserName=jsmith
  http-conf:server -
   HonorKeepAlive=true
   SuppressClientReceiveErrors=false
   SuppressClientSendErrors=false
   SendTimeout=3000
   ReceiveTimeout=3000
```

Figure 21: *Example Set of HTTP Configuration Settings in GUI*

Example 104 shows the WSDL extract that is subsequently generated for the service component of your Artix contract, based on the example settings in Figure 21 on page 238. As shown in Example 104, client and server HTTP configuration attributes are contained respectively within elements called `http-conf:client` and `http-conf:server`.

Example 104:*Extract of Example WSDL Contract*

```
<wsdl:service name="BaseService">
    <wsdl:port binding="ns1:BasePortTypeBinding" name="HTTP_Port">
        <http-conf:client Password="goofy" ReceiveTimeout="3000" SendTimeout="3000"
            URL="http://www.iona.com/support/docs/index.xml" UserName="jsmith"/>
        <http-conf:server HonorKeepAlive="true" ReceiveTimeout="3000"
            SendTimeout="3000" SuppressClientReceiveErrors="false"
            SuppressClientSendErrors="false"/>
    </wsdl:port>
</wsdl:service>
```

Adding an HTTP Port for Secure Connections

Overview

This section describes how to use the **Artix Designer** GUI to add to a WSDL contract an HTTP port that enables secure connections. It discusses the following topics:

- "SSL-related attributes" on page 240.
- "GUI steps" on page 241.
- "WSDL example" on page 241.

Note: This section deals specifically with how to set up HTTP port information within the `<service>` component of a WSDL contract. To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See the chapter relating to the payload format you are using for more details about setting up a binding for it in a WSDL contract.

SSL-related attributes

The SSL-related attributes that can be configured to be included in the `<http-conf:client>` and `<http-conf:server>` elements of an HTTP port binding are as follows:

Client SSL Attributes	Server SSL Attributes
`UseSecureSockets`	`UseSecureSockets`
`ClientCertificate`	`ServerCertificate`
`ClientCertificateChain`	`ServerCertificateChain`
`ClientPrivateKey`	`ServerPrivateKey`
`ClientPrivateKeyPassword`	`ServerPrivateKeyPassword`
`TrustedRootCertificate`	`TrustedRootCertificate`

See Table 15 on page 246 for more details of the server attributes. See Table 16 on page 253 for more details of the client attributes.

GUI steps

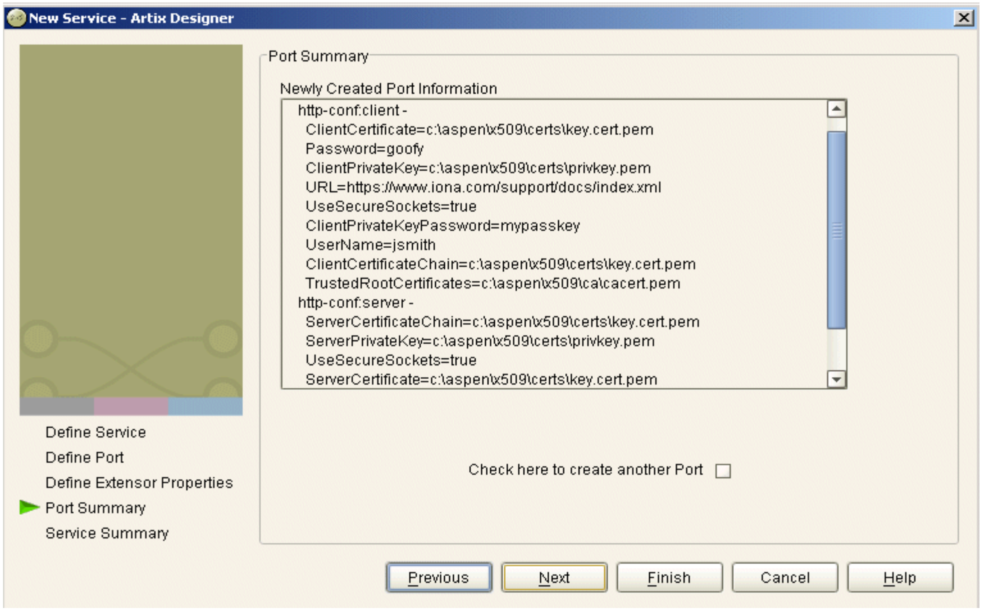
All the GUI steps described in "GUI steps" on page 234 are relevant and should be followed here, with the following stipulations:

- Specify `https://` rather than `http://` as the prefix for the value of the **URL** attribute in the **Client** configuration table.
- Enter values for the various SSL-related attributes in the **Client** and **Server** configuration tables. See "SSL-related attributes" on page 240 for a listing of these attributes. See "HTTP WSDL Extensions" on page 243 for more details about them.

Note: When you specify `https://` as the prefix for the value of the **URL** attribute in the **Client** configuration table, a secure HTTP connection is automatically enabled, even if **UseSecureSockets** is not set to `true`.

WSDL example

Figure 22 shows an example summary of SSL-related HTTP configuration settings in the GUI


New Service - Artix Designer
Port Summary
Newly Created Port Information
Define Service
Define Port
Define Extensor Properties
Port Summary
Service Summary
Check here to create another Port
Previous
Next
Finish
Cancel
Help

```
http-conf:client -
  ClientCertificate=c:\aspen\x509\certs\key.cert.pem
  Password=goofy
  ClientPrivateKey=c:\aspen\x509\certs\privkey.pem
  URL=https://www.iona.com/support/docs/index.xml
  UseSecureSockets=true
  ClientPrivateKeyPassword=mypasskey
  UserName=jsmith
  ClientCertificateChain=c:\aspen\x509\certs\key.cert.pem
  TrustedRootCertificates=c:\aspen\x509\ca\cacert.pem
http-conf:server -
  ServerCertificateChain=c:\aspen\x509\certs\key.cert.pem
  ServerPrivateKey=c:\aspen\x509\certs\privkey.pem
  UseSecureSockets=true
  ServerCertificate=c:\aspen\x509\certs\key.cert.pem
```

Figure 22: *Example Set of SSL-Related HTTP Configuration Settings*

Example 105 shows the WSDL extract that is subsequently generated for the service component of your Artix contract, based on the example settings in Figure 22 on page 241. As shown in Example 105, client and server HTTP configuration attributes are contained respectively within elements called `http-conf:client` and `http-conf:server`.

Example 105:*Extract of Example WSDL Contract with SSL Attributes*

```
<wsdl:service name="BaseService">
    <wsdl:port binding="ns1:BasePortTypeBinding" name="HTTP_SSL_Port">
        <http-conf:client ClientCertificate="c:\aspen\x509\certs\key.cert.pem"
                          ClientCertificateChain="c:\aspen\x509\certs\key.cert.pem"
                          ClientPrivateKey="c:\aspen\x509\certs\privkey.pem"
                          ClientPrivateKeyPassword="mykeypass" Password="goofy"
                          TrustedRootCertificates="c:\aspen\x509\ca\cacert.pem"
                          URL="https://www.iona.com/support/docs/index.xml"
                          UseSecureSockets="true"
                          UserName="jsmith"/>
        <http-conf:server ServerCertificate="c:\aspen\x509\certs\key.cert.pem"
                          ServerCertificateChain="c:\aspen\x509\certs\key.cert.pem"
                          ServerPrivateKey="c:\aspen\x509\certs\privkey.pem"
                          ServerPrivateKeyPassword="mykeypass"
                          TrustedRootCertificates="c:\aspen\x509\ca\cacert.pem"
                          UseSecureSockets="true"/>
    </wsdl:port>
</wsdl:service>
```

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 106 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 106:*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="…"
                          CacheControl="…"
                          Cookie="…"
                          BrowserType="…"
                          Referer="…"
                          ProxyServer="…"
                          ProxyUserName="…"
                          ProxyPassword="…"
                          ProxyAuthorizationType="…"
                          ProxyAuthorization="…"
                          UseSecureSockets="…"
```

Example 106:*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 246.
- "Client configuration attributes" on page 253.

Server configuration attributes

Table 15 describes the server-side configuration attributes for the HTTP transport that are defined within the `http-conf:server` element.

Table 15: *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 `3000`.
`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 `3000`.
`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 15: *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 15: *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 15: *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 15: *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 15: *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 15: *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 16 describes the client-side configuration attributes for the HTTP transport that are defined within the `http-conf:client` element.

Table 16: *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 `3000` (that is, 30 seconds).
`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 `3000` (that is, 30 seconds).
`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 16: *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 16: *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 16: *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 16: *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.

Table 16: *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 16: *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 16: *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 16: *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 16: *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 16: *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 243 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 243.

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

Table 17: *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 17: *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 258 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 17 describes the transport attributes that can be sent from a server to a client in the header of a response message.

Table 18: *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 248 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

Using the WebSphere MQ Plug-in

The Artix WebSphere MQ plug-in and the associated WSDL binding extensions provides the ability to integrate with 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.

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 19 shows the matrix of WebSphere MQ features Artix supports.

Table 19: *Supported WebSphere MQ Features*

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

The Artix Designer walks you through adding an WebSphere MQ port to an Artix contract and ensures that you include all of the required information. If you are comfortable editing Artix contracts, you can also describe the port manually using any standard text editor.

WebSphere MQ port elements

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

<mq:client> describes the port Artix client applications use to connect to a WebSphere MQ server application.

<mq:server> describes the port WebSphere MQ client applications use to connect to Artix.

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

In this section

This section discusses the following topics:

Configuring an Artix WebSphere MQ Port

Overview

The Artix WebSphere MQ port description is specified by the namespace `http:\\schemas.iona.com\transports\mq`. It defines a number of attributes to configure an WebSphere MQ port. Table 20 lists the defined attributes. They are described in detail following the table.

Table 20: *WebSphere MQ Port Attributes*

Attributes	Description
`QueueManagerName`	Specifies the name of the queue manager.
`QueueName`	Specifies the name of the message queue.
`ReplyQueueName`	Specifies the name of the queue where response messages are received.
`ReplyQueueManager`	Specifies the name of the reply queue manager.
`ModelQueueName`	Specifies the name of the queue to be used as a model for creating dynamic queues.
`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 the type of identifier to be used to correlate request and response messages with each other.
`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 how long messages are retained in the queue.
`MessagePriority`	Specifies the priority with which messages will be processed.
`DeliveryMode`	Specifies the delivery mode of the messages sent to the queue.

Table 20: *WebSphere MQ Port Attributes*

Attributes	Description
`Transactional`	Specifies if transaction operations must be performed on the messages sent to the queue.
`ReportOption`	Specifies how the queue reports message activity.
`FormatType`	Specifies what type of data is contained in the message body.
`MessageId`	Specifies a unique ID to assist in correlating messages with their responses.
`CorrelationId`	Specifies a unique ID to assist in correlating messages with their responses.
`ApplicationData`	Specifies optional information to be associated with the message.
`AccountingToken`	Specifies user-supplied information for accounting purposes.
`Convert`	Specifies in the messages in the queue need to be converted to the system's native encoding.

QueueManagerName

`QueueManagerName` specifies the name of the WebSphere MQ queue manager that controls the message queue the port uses. Defaults to the local queue manager name.

QueueName

`QueueName` is a required attribute for an WebSphere MQ port. It specifies the message queue the port uses.

ReplyQueueName

`ReplyQueueName` specifies the name of the reply message queue used by the port.

ReplyQueueManager

`ReplyQueueManager` specifies the name of the WebSphere MQ queue manager that controls the reply message queue. Defaults to the local queue manager name.

ModelQueueName

`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.

ConnectionName

`ConnectionName` is a required attribute for an Artix WebSphere MQ port. It specifies the name of the connection Artix uses to connect to its queue.

ConnectionReusable

`ConnectionReusable` specifies if the connection named in the `ConnectionName` field can be used by more than one application. Valid entries are yes and no. Defaults to no.

ConnectionFastPath

`ConnectionFastPath` specifies if you want to load the queue manager in process. Valid entries are yes and no. Defaults to no.

UsageStyle

`UsageStyle` specifies if a message can be queued without expecting a response. Valid entries are peer, requester, and responder as described in Table 21.

Table 21: *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.

CorrelationStyle

`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`.

Table 22 shows the actions of MQGET and MQPUT when receiving a message using specified message ID and correlation ID.

Table 22: *MQGET and MQPUT Actions*

Artix Port Setting	Action for MQGET	Action for MQPUT
messageId	Set correlation ID on message descriptor to message ID	Copy message ID onto message descriptor's Correlation_ID
correlationId	Set correlation ID on message descriptor to CorrelationID	Copy CorrelationID onto message descriptor's CorrelationID
messageId copy	Set message ID on message descriptor to messageID	Copy MessageID onto message descriptor's MessageID

AccessMode

AccessMode is a required attribute for an Artix WebSphere MQ port. It controls the action of MQOPEN in the Artix WebSphere MQ transport. Its values can be peek, send, recive, receive exclusive, and receive shared, as explained in Table 23.

Table 23: *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 so that it is open to receive messages.
receive	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).

Table 23: *Artix WebSphere MQ Access Modes*

Attribute Setting	Description
`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.
`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`.

Timeout

`Timeout` specifies the amount of time, in milliseconds, that a message can sit on the queue 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.

MessageExpiry

`MessageExpiry` 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.

MessagePriority

MessagePriority defines the message 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).

DeliveryMode

DeliveryMode can be persistent or not persistent. persistent means that the message survives both system failures and restarts of the queue manager. Internally, this sets MQMD Persistence of the Artix WebSphere MQ port to MQPER_PERSISTENT or MQPER_NOT_PERSISTENT. To support transactional messaging, you must make the messages persistent.

Transactional

Transactional controls the ability for a message to participate in a transaction. Valid values are yes and no. For a yes value, messages operate within the normal unit-of-work protocols; a message is not visible outside the unit of work until the unit of work is committed. If the unit of work is rolled back, the message is deleted from the queue. For a no value, messages operate outside the normal unit-of-work protocols; a message is available immediately and it cannot be deleted by rolling back a unit of work.

The default value is no.

ReportOption

ReportOption 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. The values of this attribute are explained in Table 21.

Table 24: *ReportOption Attribute Settings*

Attribute Setting	Description
none (Default)	Corresponds to MQRO_NONE. none specifies that no reports are required. This value can be used to indicate that no other options have been specified.

Table 24: *ReportOption Attribute Settings*

Attribute Setting	Description
`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.
`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

FormatType

`FormatType` 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

The special values for `Format` can be `none`, `string`, `event`, `programmable command`, and `unicode` , as described in Table 25.

Table 25: *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.
`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.

MessageId	`MessageId` is an alphanumeric string of up to 20 bytes in length. 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.
CorrelationId	`CorrelationId` is an alphanumeric string of up to 20 bytes in length. 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.
ApplicationData	`ApplicationData` specifies any application specific information that needs to be set in the message header.
AccountingToken	`AccountingToken` specifies application specific information used for accounting purposes.
Convert	`Convert` specifies if messages are to be converted to the receiving systems native data format. Valid values are yes and no. Default is no.

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> describes the port Artix client applications use to connect to an WebSphere MQ server application.

<mq:server> describes the port WebSphere MQ client applications use to connect to Artix.

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.

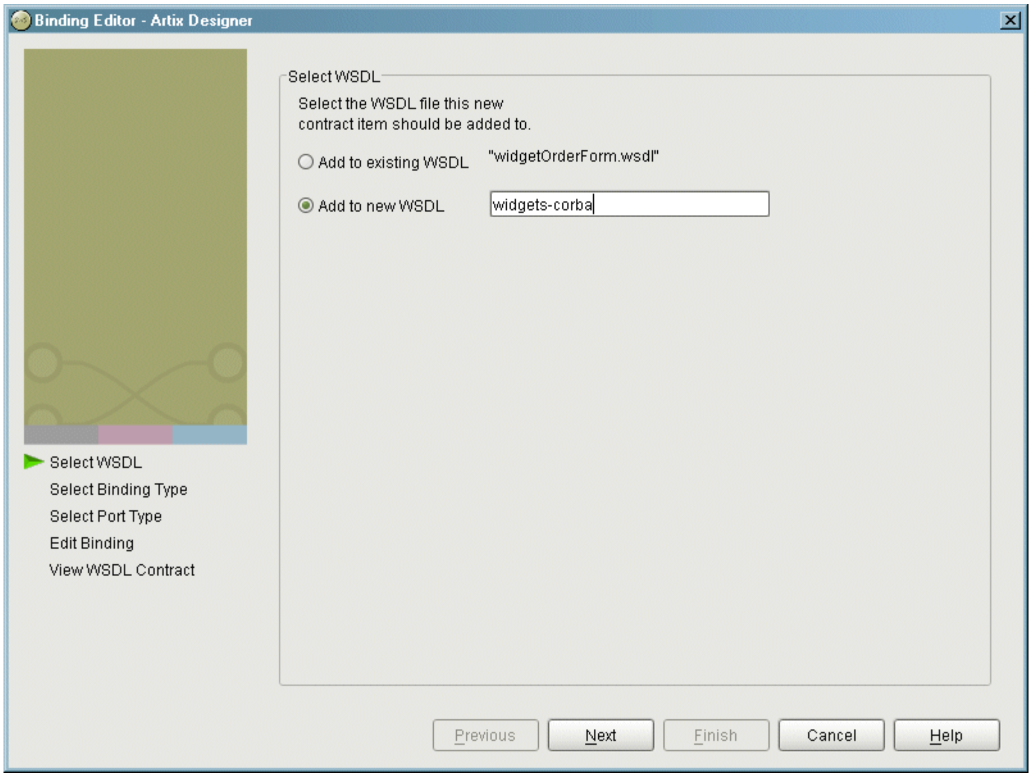
Artix Designer walks you through the process of adding an WebSphere MQ port to an Artix contract.

Procedure

To add an WebSphere MQ port to an Artix contract complete the following steps:

1. Select the node for the service to which you want to add the WebSphere MQ port from the project tree.
2. Select **Contracts|New|Service** from the Designer menu.

3. You will see a screen like Figure 23.



Select WSDL

Select the WSDL file this new contract item should be added to.

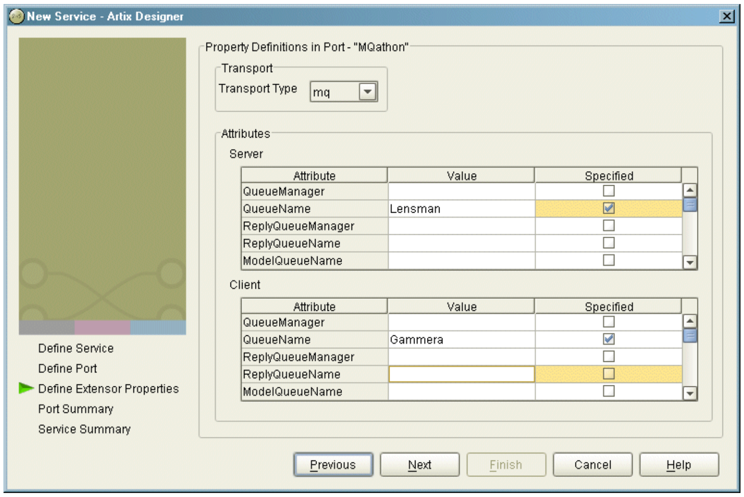
( ) Add to existing WSDL "widgetOrderForm.wsdl"

(●) Add to new WSDL widgets-corba

Figure 23: *Select WSDL location*

4. Select where to create the WSDL entry for the new service.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Enter the name for the new service.

7. Click **Next**.
8. Enter a name for the new port.
9. Select the desired binding from the **Available Bindings** pull-down list.
10. Click **Next**.
11. Select **mq** from the **Transport Type** pull-down list.
12. You will see a screen like Figure 24.



Property Definitions in Port - "MQathon"

Transport

Transport Type: mq

Attributes

Server

Attribute	Value	Specified
QueueManager		☐
QueueName	Lensman	☑
ReplyQueueManager		☐
ReplyQueueName		☐
ModelQueueName		☐

Client

Attribute	Value	Specified
QueueManager		☐
QueueName	Gammera	☑
ReplyQueueManager		☐
ReplyQueueName		☐
ModelQueueName		☐

Figure 24: *WebSphere MQ Port Properties*

13. Enter values for the desired attributes.

    You must supply values for the `QueueName` and `AccessMode` of the port at a minimum.
14. Ensure that the attributes you want set have a check mark in the **Specified** column.
15. Click **Next**.
16. Click **Finish**.

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 13

Using the Tuxedo Plug-in

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 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.

ATMI support

Artix supports the following ATMI features:

Table 26: *Artix ATMI Feature Support*

Feature	Supported	Not Supported

FML support

Artix supports the following FML features:

Table 27: *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

Table 27: *Artix FML Feature Support*

Feature	Supported	Not Supported
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 108.

Example 108:*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 109. 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 109:*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 110.

Example 110:*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 108 on page 289, will be similar to the binding shown in Example 111.

Example 111:*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.

Describing a Tuxedo port

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 namespace:

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

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

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>` takes a single mandatory attribute, `serviceName`, which specifies the bulletin board name of the Tuxedo port being exposed.

Using Artix Designer to add a Tuxedo port to an Artix contract

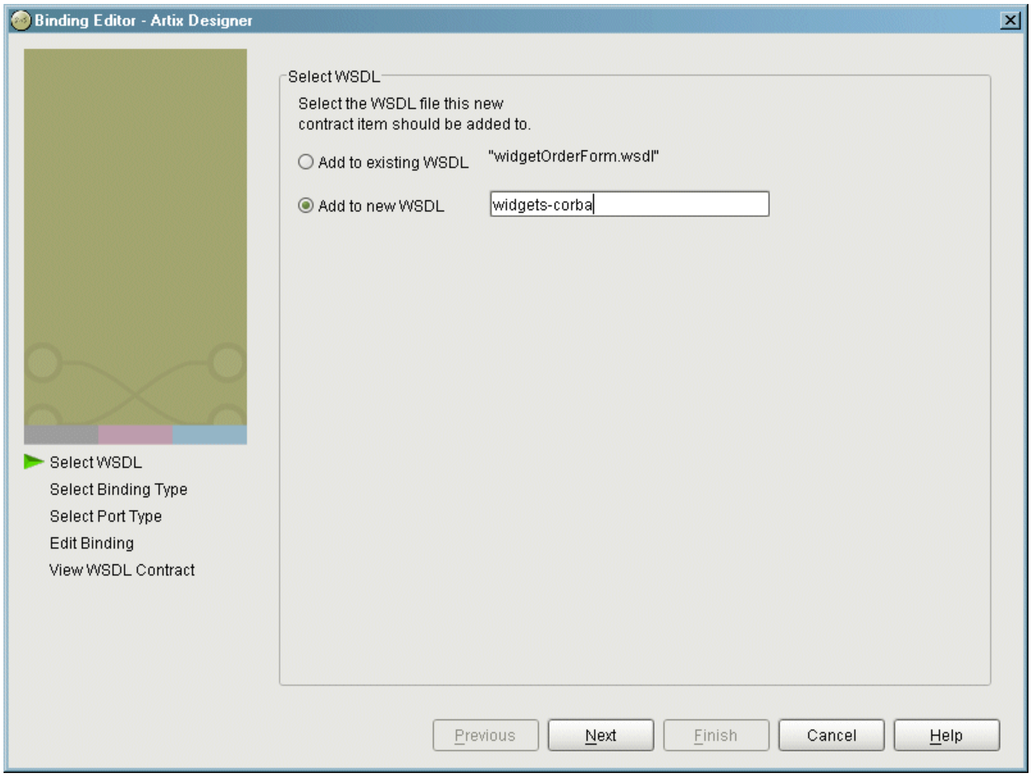
To add a Tuxedo port to an Artix contract using Artix Designer complete the following steps:

1. Select the contract you to which you are going to add the Tuxedo port.

   Note: The contract must have an existing SOAP binding before Artix Designer will allow you to add a Tuxedo port to the contract.

2. Select **Services|New Service...** from the **Contract** menu.

3. You will see a screen like Figure 25.



Select WSDL

Select the WSDL file this new contract item should be added to.

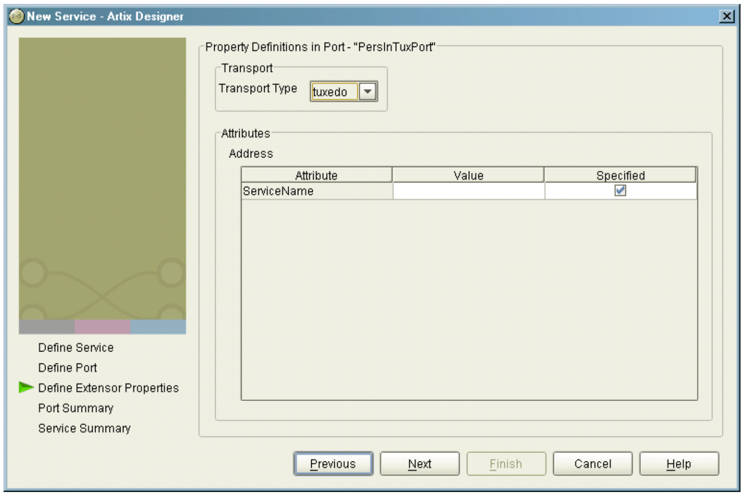
Add to existing WSDL "widgetOrderForm.wsdl"

[✓ Add to new WSDL] widgets-corba

Figure 25: *Select WSDL Location*

4. Select where to create the WSDL entry for the new service.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Enter a new for the new service and click **Next**.

7. Enter a name for the Tuxedo port.
8. From the **Binding** drop-down list select the SOAP binding which this port will expose to the network.
9. Click **Next**.
10. Select **tuxedo** from the **Transport** drop-down list.
11. The **Attributes** table will look similar to the one shown in Figure 26.



Property Definitions in Port - "PersInTuxPort"

Transport

Transport Type: tuxedo

Attributes

Address

Attribute	Value	Specified
ServiceName		✓

Figure 26: *Edit Tuxedo Port Properties*

12. Enter a valid Tuxedo service name in the ServiceName **Value** field.
13. Click **Next** to review the settings for the new service and Tuxedo port.
14. Click **Finish** to create the new service and Tuxedo port.

Artix Designer will create a new contract containing the new service and place it in the project tree.

Example

An Artix contract exposing the `personalInfoService`, defined in Example 111 on page 292, would contain a `<service>` element similar to Example 112 on page 296.

Example 112:*Tuxedo port description*

```
<service name="personalInfoService">
  <port binding="tns:personalInfoBinding" name="tuxInfoPort">
    <tuxedo:server serviceName="personalInfoService" />
  </port>
</service>
```

Embedding Artix in the Tuxedo Container

Overview

In order to have Artix interact properly with native Tuxedo applications, you need to embed Artix into the Tuxedo container. At a minimum this involves adding information about Artix to your Tuxedo configuration file and registering your Artix processes with the Tuxedo bulletin board. You can also have Tuxedo bring up your Artix process as a Tuxedo server when running tmboot.

Procedure

To embed an Artix process into a Tuxedo container complete the following steps:

1. Ensure that your environment is properly configured for Tuxedo.
2. Add the Tuxedo plug-in, tuxedo, to your Artix process's orb_plugins list. See "ORB Plug-ins List" on page 36.

```
orb_plugins=["iiop_profile", "giop", "iiop", "tuxedo"];
```

3. Set plugins:tuxedo:server to true in your Artix configuration scope.
4. Ensure that the executable for your Artix process is placed into the directory specified in the APPDIR entry of your Tuxedo configuration.
5. Edit your Tuxedo configuration's SERVERS section to include an entry for your Artix process.

   For example, if the executable of your Artix process is boingo, you make the following entry in the SERVERS section:

```
boingo SVRGRP=OINGO SVRID=1
```

   This associates boingo with the Tuxedo group called OINGO in your configuration and assigns boingo a server ID of 1. You can modify the server's properties as needed.
6. Edit your Tuxedo configuration's SERVICES section to include an entry for your Artix process.

   While standard Tuxedo servers only require a SERVICES entry if you are setting optional runtime properties, Artix servers in the Tuxedo container require an entry even if no optional runtime properties are being set. The name entered for the Artix process is the name specified in the `serviceName` attribute of the Tuxedo port defined in the process' Artix contract.

For example, given the port definition shown in Example 112 on page 296, the `SERVICES` entry would be `personalInfoService`.

7. If you made the Tuxedo configuration changes in the ASCII version of the configuration, `UBBCONFIG`, reload the `TUXCONFIG` with `tmload`.

Once you have properly configured Tuxedo, it will manage your Artix process as if it were a regular Tuxedo server.

Using the TIBCO Rendezvous Plug-in

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.

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 28 shows the matrix of TIBCO Rendezvous features Artix supports.

Table 28: *Supported TIBCO Rendezvous Features*

Feature	Supported	Not Supported
Server Side Advisory Callbacks	x	
Certified Message Delivery	x	
Fault Tolerance (`TibrvFtMember/Monitor`)		x
Virtual Connections (`TibrvVcTransport`)		x
Secure Daemon (`rvsd/TibrvSDContext`)		x
`TIBRVMSG_IPADDR32`		x
`TIBRVMSG_IPPORT16`		x

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 29.

Table 29: *TibrvMsg Binding Attributes*

Attribute	Description
`tibrv:binding`	Specifies that the interface is exposed using TibrvMsgs.
`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 output 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.
`tibrv:output@messageNameFieldValue`	Specifies the field value that corresponds to the output message name.

TIBRVMSG type mapping

Table 30 shows how TibrvMsg data types are mapped to XSD types in Artix contracts and C++ data types in Artix application code.

Table 30: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD	Artix C++
TIBRVMSG_STRING1	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_DATETIME2	xsd:dateTime	IT_BUS::DateTime
TIBRVMSG_OPAQUE	xsd:base64Binary	IT_BUS::Base64Binary
TIBRVMSG_OPAQUE	xsd:hexBinary	IT_BUS::HexBinary
TIBRVMSG_MSG3	xsd:complexType/sequence	IT_BUS::SequenceComplexType
TIBRVMSG_MSG4	xsd:complexType/all	IT_BUS::AllComplexType
TIBRVMSG_MSG5	xsd:complexType/choice	IT_BUS::ChoiceComplexType
TIBRVMSG_*ARRAY/MSG6	xsd:complexType/sequence with element MaxOccurs > 1	IT_BUS::Array

Table 30: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD	Artix C++
`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.

Example

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.

Port attributes

Table 31 lists the Artix contract elements used to describe a TIB/RV port.

Table 31: *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@stringEncoding`	Specifies the charset used to encode `TIBRVMSG_STRING` data.
`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.

Table 31: *TIB/RV Transport Properties*

Attribute	Explanation
`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.
`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.

Table 31: *TIB/RV Transport Properties*

Attribute	Explanation
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 32.

Table 32: *TIB/RV Supported Payload formats*

Setting	Payload Formats	TIB/RV Message Implications
msg	TibrvMsg	The top-level messages will have a field of TIBRVMSG_STRING, a null name, and an ID of 0. This field will contain the name of the operation, from the Artix contract, that should be invoked.
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@stringEncoding	`tibrv:port@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. **Note:** The infrastructure will not perform any charset conversions, and this is purely for the contract between client and server application implementation.
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`.

CHAPTER 15

Using the IIOP Tunnel

The IIOP tunnel provides 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.

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

Modifying a Contract to Use the IIOP Tunnel

Overview

Service Access Points (SAPs) that use the IIOP tunnel 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, 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 "Using the CORBA Naming Service" on page 321.

- 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 the name of the POA 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" />
```

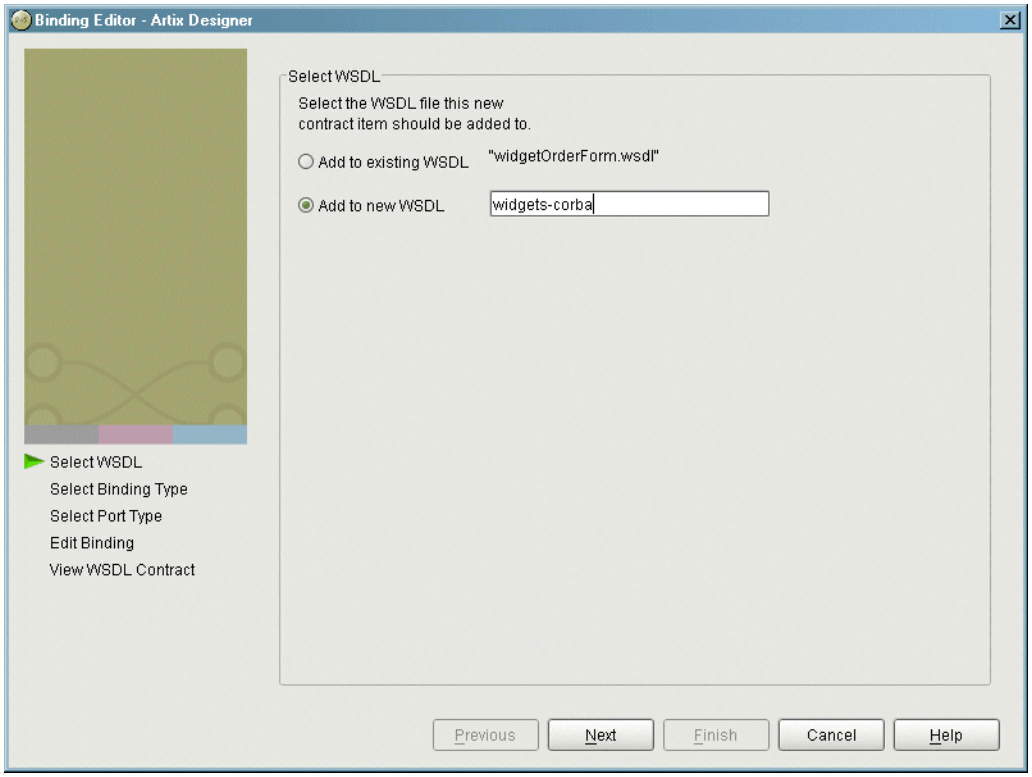
This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Procedure

To add an IIOP tunnel port to your service contract using the GUI, complete the following steps:

1. From the project tree, select the contract to which you want to add the IIOP tunnel port.
2. Select **Services|New Service** from the **Contract** menu of the designer.

3. You will see a screen like Figure 27.


Binding Editor - Artix Designer

Select WSDL

Select the WSDL file this new contract item should be added to.

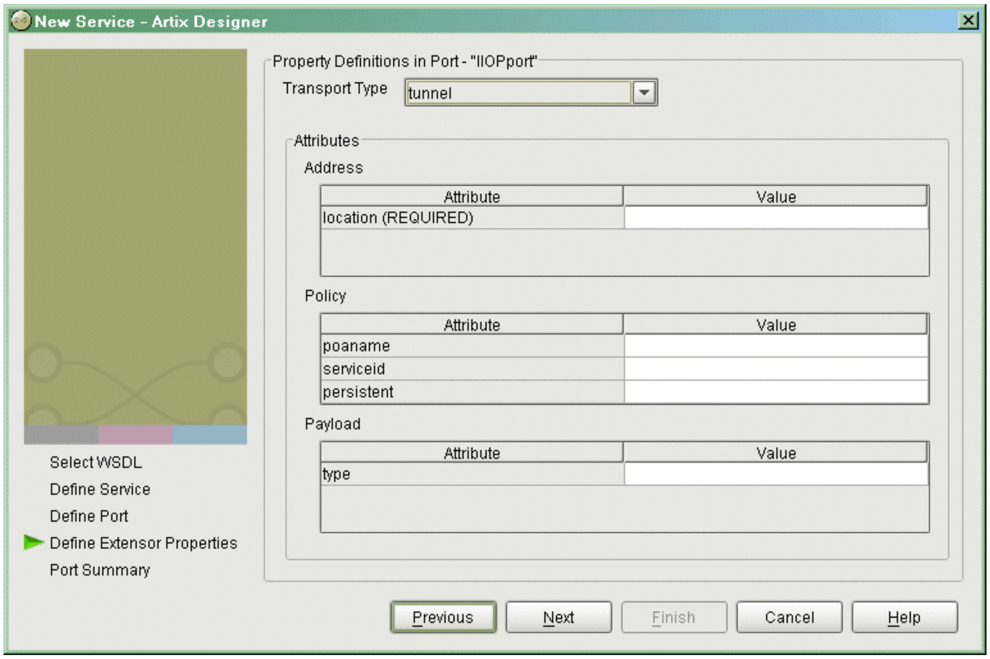
( ) Add to existing WSDL "widgetOrderForm.wsdl"

(●) Add to new WSDL widgets-corba

Figure 27: *Select WSDL Location*

4. Select where to create the WSDL entry for the new service.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Enter a unique name for the new service.

7. Click **Next**.
8. Enter a name for the new IIOP tunnel port that is being created.
9. From the drop down list, select the binding that the port is going to expose.
10. Click **Next**.
11. You will see a dialog similar to Figure 28.



Property Definitions in Port - "IIOPport"

Transport Type: tunnel

Attributes

Address

Attribute	Value
location (REQUIRED)	

Policy

Attribute	Value
poaname	
serviceid	
persistent	

Payload

Attribute	Value
type	

Figure 28: *Edit IIOP Tunnel Port Properties*

12. From the drop down list in the **Transport** box, select **tunnel**.
13. In the **Address** table, enter the address in the line for **Location**.
14. If you want to set any of the supported POA policies, place a check in the **Specified** box on the appropriate line in the **Policy** table and enter a valid value.
15. Click **Next**.

16. Review the settings for the new IIOP tunnel port.
17. If it is correct, click **Next**.
18. Review the settings for the new service in which the IIOP port is described.
19. If it is correct, click **Finish**.

Example

For example, an IIOP tunnel port for the `personalInfoLookup` binding would look similar to Example 113:

Example 113:*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.

Using the CORBA Naming Service

Overview

In order to fully integrate with deployed CORBA systems, Artix can use a CORBA naming service that supports the `CosNaming` interface. Doing so requires editing the port information in the service's contract and modifying the Artix configuration.

Servers

To specify that an Artix instance (acting as proxy for a server) is to use the COBRA naming service, you edit the `<iiop:address>` element of the IIOP tunnel port. In place of the file name used in the `location` attribute, specify a `corbaname`. For example, to specify that the IIOP tunnel for the personal info server publishes its IOR to the CORBA naming service, specify the `<corba:address>` as follows:

```
<corba:address location="corbaname:rir:/NameService#personalInfoService"/>
```

This registers the server in the name service under the name `personalInfoService`.

Clients

An Artix instance (acting as a proxy for a client) can also use the `<iiop:address>` element to specify what name to look up in the CORBA name service. The name the client looks up in the name service is the string after the `#` in the specified location. For example, a client using the `<iiop:address>` shown above in "Servers" looks up the IOR for an object named `personalInfoService`.

Configuration

Artix applications that wish to use a CORBA name service must be configured to load a name resolver plug-in and have an initial reference for the running name service.

To modify the Artix configuration do the following:

1. Open the Artix configuration file, `IT_PRODUCT_DIR\artix\1.2\etc\artix.cfg`, in a text editor.
2. In the global scope, add the following lines:

```
initial_references:NameService:reference="corbaloc::localhost:portNumber/NameService";
url_resolvers:corbaname:plugin="naming_resolver";
plugins:naming_resolver:shlib_name="it_naming";
```

`portNumber` is the number of the port on which the name service is running.

For more information on Artix configuration, see "Configuration" on page 27.

Payload Formats

Artix supports several transport independent payload format such as SOAP and Fixed Record Length buffers.

In this chapter

This chapter discusses the following topics:

G2++ Data Format

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 114:*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 115:*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 115:*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 116:*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 at 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 defines 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 117:*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 118:*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 119: *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 120:*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.

Fixed Record Length Data Format

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.

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.

In this section

This section discusses the following topics:

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>

<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. It takes one opetional attribute, `discriminator`, which allows you to assign a name to the operation for identifying the operation as it is sent down the wire by the Artix runtime.

<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 Fixed String MessageExample 121 would be mapped to a `<fixed:field>` similar to Example 122.

Example 121:*Fixed String Message*

```
<message name="fixedStingMessage">
 <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 122:*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 123.

Example 123:*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 124. This provides Artix with the meta-data needed to properly handle the data.

Example 124:*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 125.

Example 125:*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 126.

Example 126:*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 125 on page 335, were mapped the the fixed field shown in Example 127, the actual message returned from the binding would always have the date 11/11/2112.

Example 127:*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 128.

Example 128:*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 128 could be mapped to the fixed field shown in Example 129. Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to `OranceTango`.

Example 129:*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 must also use counterName. counterName specifies the field used for indexing the array or sequence. The value of counterName corresponds to a bindingOnly <fixed:field> with at least enough digits to count to the value specified in occurs as shown in Example 130.

Example 130:*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 131 shows an Artix contract fragment for such a mapping.

Example 131:*Mapping a Sequence to a Fixed Record Length Message*

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

Example 131: *Mapping a Sequence to a Fixed Record Length Message*

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

<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 132. 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 132 must be two digit integers.

Example 132:*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 unions 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:chioce>`.

`<fixed:case>` must contian 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 133 shows an Artix contract fragment mapping a union to a fixed record length message.

Example 133:*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/">
```

Example 133:*Mapping a Union to a Fixed Record Length Message*

```
   <xsd:complexType name="unionStationType">
     <xsd:choice>
       <xsd:element name="train"  type="xsd:string"/>
       <xsd:element name="bus"    type="xsd:int"/>
       <xsd:element name="cab"    type="xsd:int"/>
       <xsd:element name="subway" type="xsd:string" />
     </xsd:choice>
   </xsd:complexType>
...
</types>
<message name="fixedSequence">
  <part name="stationPart" type="tns:unionStationType" />
</message>
<portType name="fixedSequencePortType">
...
</portType>
<binding name="fixedSequenceBinding"
         type="tns:fixedSequencePortType">
  <fixed:binding />
...
    <fixed:field name="disc" format="##" bindingOnly="true" />
    <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>
```

Adding a Fixed Record Length Binding to an Artix Contract

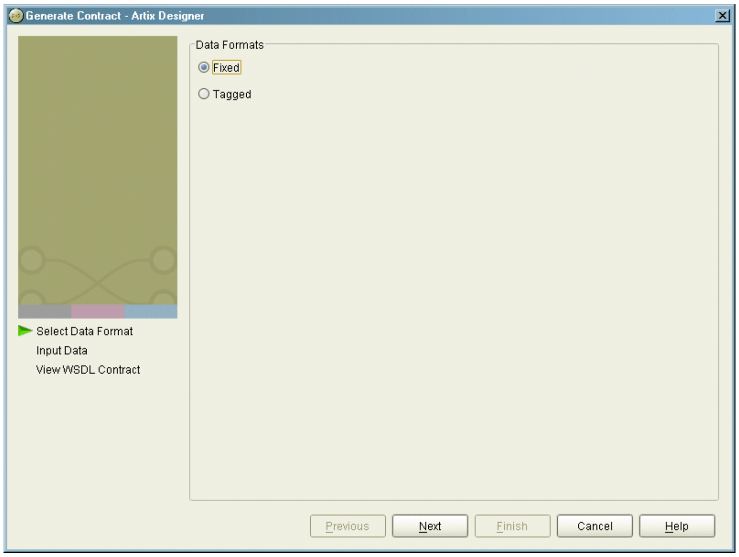
Overview

Currently Artix does not provide an automated tool to generate fixed record length message bindings for logical interfaces defined in an Artix contract. You must hand enter the mapping information or create a new contract in Artix Designer using the fixed record length data description as a starting point.

Using Artix Designer

To create a new contract using fixed record length data complete the following steps:

1. Select **New|Contract From...**.
2. You will see a screen similar to Figure 29.

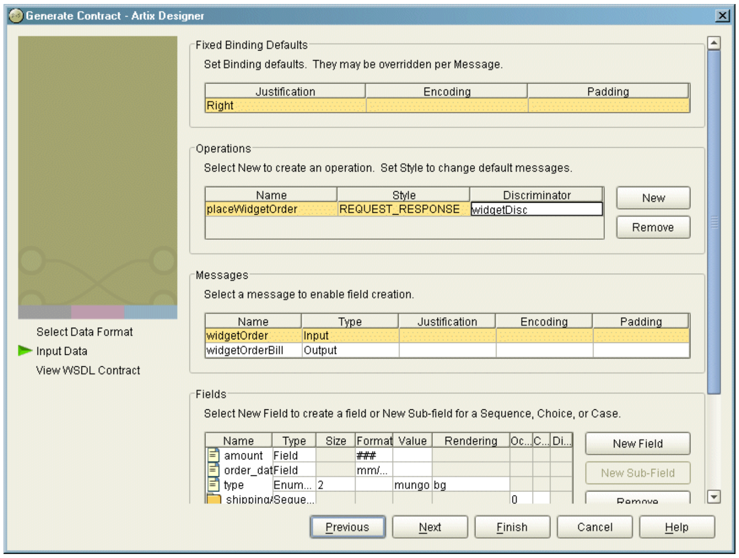


Data Formats
[✓ Fixed]
Tagged

Figure 29: *Binding Selection*

3. Select **Fixed**.
4. Click **Next** to enter the binding information.

5. You will see a screen similar to Figure 30.



Fixed Binding Defaults

Set Binding defaults. They may be overridden per Message.

Justification	Encoding	Padding
Right		

Operations

Select New to create an operation. Set Style to change default messages.

Name	Style	Discriminator
placeWidgetOrder	REQUEST_RESPONSE	widgetDisc

Messages

Select a message to enable field creation.

Name	Type	Justification	Encoding	Padding
widgetOrder	Input			
widgetOrderBill	Output			

Fields

Select New Field to create a field or New Sub-field for a Sequence, Choice, or Case.

Name	Type	Size	Format	Value	Rendering	Oc...	C...	Di...
amount	Field		###					
order_dat	Field		mm/...					
type	Enum...	2		mungo	bg			
shipping/	Seque...					0		

Figure 30: *Fixed Binding Information Screen*

6. Under the **Fixed Bindings Defaults** enter the default justification, encoding, padding for this binding.

   These values correspond to the `justification`, `encoding`, and `padHexCode` attributes of the `<fixed:binding>` tag as described on page 332.

7. Under **Operations** enter the information for the operations your service offers.

8. Under **Messages** enter the messages for the operation selected in the **Operations** field.

   You are able to provide alternate values for the `justification`, `encoding`, and `padHexCode` attributes here. These values are set on the `<fixed:body>` tag as described on page 333.

9. Under **Fields** enter the fields that make up the message selected in the **Messages** field.

Each message part can be either a field as described in "<fixed:field>" on page 333, an enumeration as described in "<fixed:enumeration>" on page 336, a sequence as described in "<fixed:sequence>" on page 337, or a choice as described in "<fixed:choice>" on page 338.

10. Click **Finish** to create the contract with the fixed record binding.

Example

Example 134 shows an example of an Artix contract containing a fixed record length message binding.

Example 134:*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://schames.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/">
      <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>
```

Example 134:*Fixed Record Length Message Binding*

```
      <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>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
  </portType>
```

Example 134:*Fixed Record Length Message Binding*

```
<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 134:*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>
```

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:fixed="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 415 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 135 can be mapped to an XML format binding:

Example 135:*Valid XML Binding Message*

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

However, the message in Example 136 cannot be mapped to an XML format binding because it has more than one part.

Example 136:*Invalid XML Binding Message*

```
<message name="matilldas">
  <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 137 shows how the widget service would be mapped to an XML format binding.

Example 137:*XML Format Binding for Widgets*

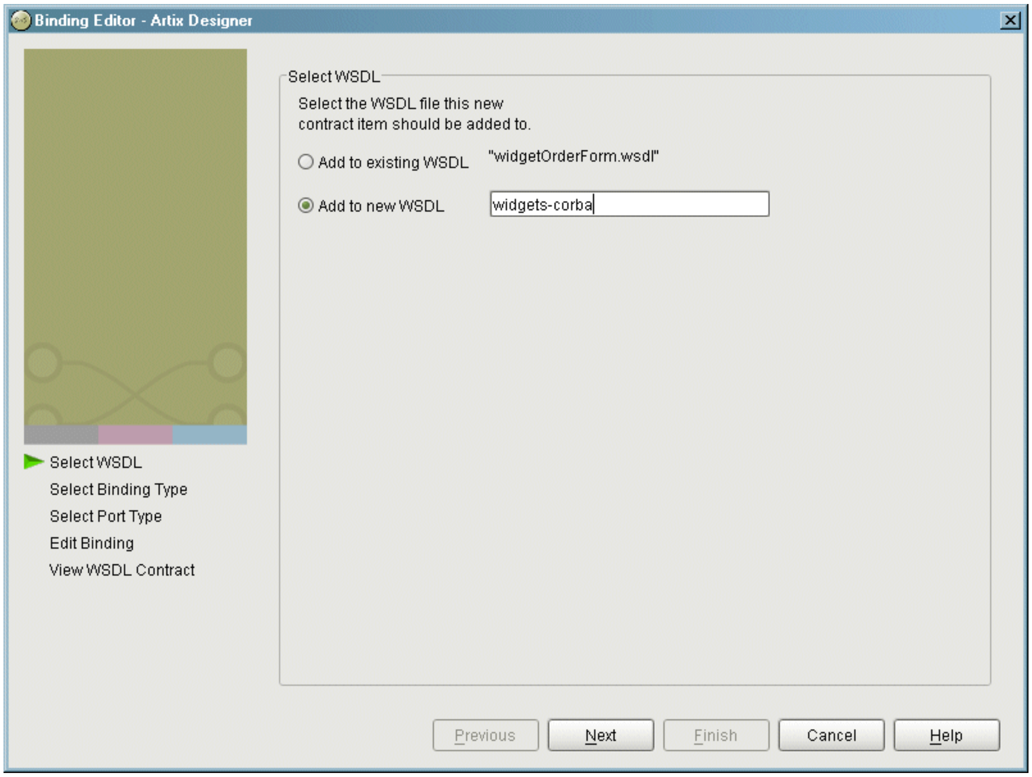
```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input 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>
```

Adding an XML format binding to an Artix Contract

To add an XML format binding to an Artix contract using Artix designer complete the following steps:

1. From the project tree, select the service to which you want to add the XML format binding.
2. Select **Contract|Bindings|New Binding** from the menu of the designer.

3. You will see a screen like Figure 31.



Select WSDL

Select the WSDL file this new contract item should be added to.

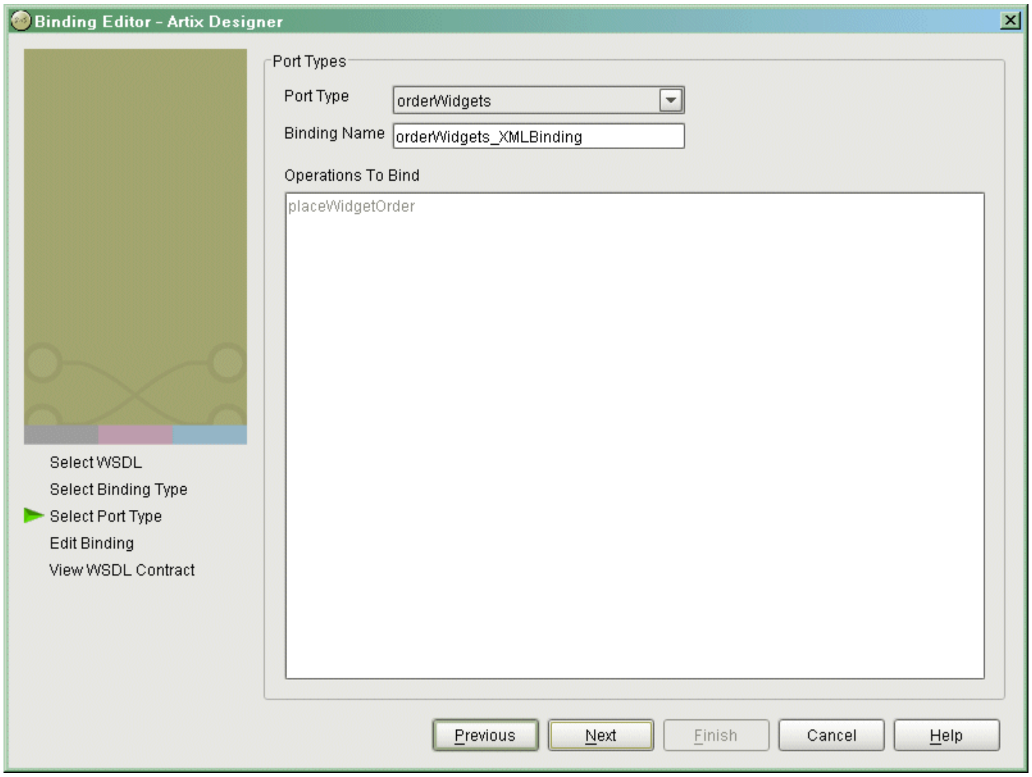
( ) Add to existing WSDL "widgetOrderForm.wsdl"

[✓ Add to new WSDL] widgets-corba

Figure 31: *Select WSDL location*

4. Select where to create the WSDL entry for the new binding.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Select **XML** from the list of possible bindings.

7. Click **Next** to select the interface you want mapped to the XML format binding.
8. You will see a dialog similar to Figure 32.



Port Types

Port Type: orderWidgets

Binding Name: orderWidgets_XMLBinding

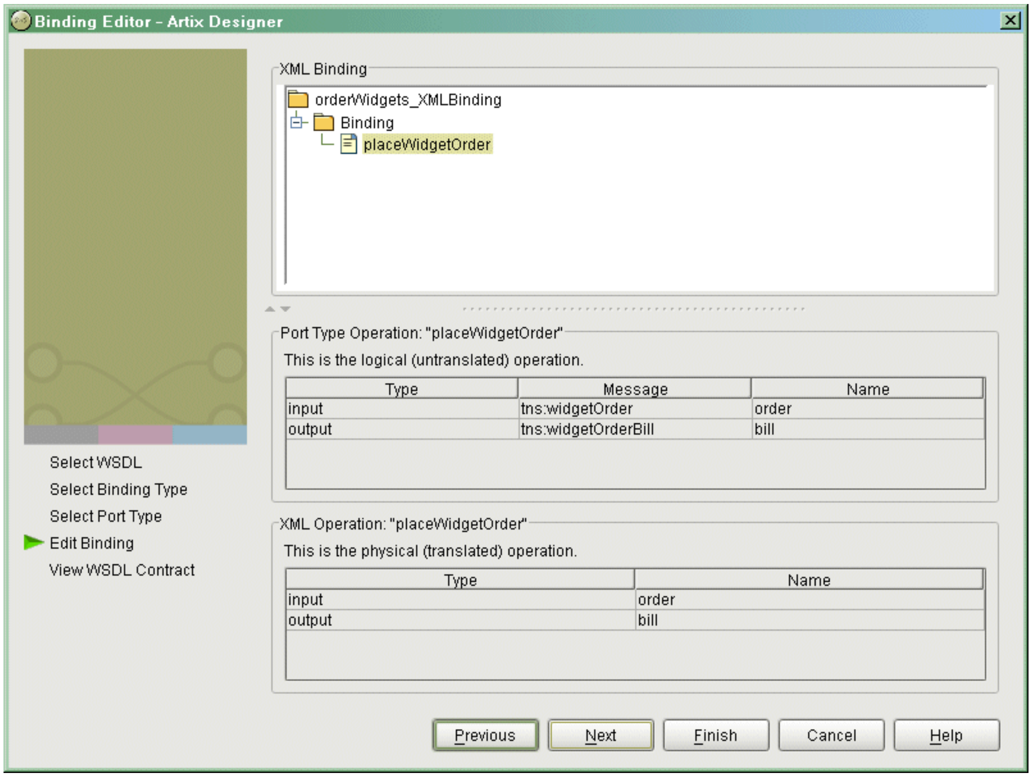
Operations To Bind

placeWidgetOrder

Figure 32: *Select Interface to Map to XML Format*

9. From the drop down list select the interface you want to map to the XML format binding.
10. Enter the name for the new binding.
11. If there is more than one operation described in the interface, select the operation that are to be mapped into the XML format binding.
12. Click **Next** to edit the new XML format binding.

13. You will see a dialog similar to Figure 33.



XML Binding

orderWidgets_XMLBinding
Binding
placeWidgetOrder

Port Type Operation: "placeWidgetOrder"

This is the logical (untranslated) operation.

Type	Message	Name
input	tns:widgetOrder	order
output	tns:widgetOrderBill	bill

XML Operation: "placeWidgetOrder"

This is the physical (translated) operation.

Type	Name
input	order
output	bill

Figure 33: *Edit the CORBA Binding*

14. Examine the different elements of the binding by selecting them from the tree at the top of the dialog.
15. Edit the values shown in white if they are not correct.
16. When you are finished editing the binding, click **Next**.
17. Review the newly created contract containing the new XML format binding.
18. If the contract is correct, click **Finish**.

Tagged Data Format

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.

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.

In this section

This section 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>

<tagged:binding>

`<tagged:binding>` specifies that the binding is for tagged data format messages. It has five 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`(,), 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`.

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.

<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 taged data on the wire and describes the exact mapping for the message's parts.

`<tagged:body>` takes six optional attributes:

`selfDescribing`	Specifies if the message data on the wire includes the field names. Valid values are `true` or `false`. If this attribute is set to `false`, the setting for `fieldNameValueSeparator` is ignored.
`fieldSeparator`	Specifies the delimiter the message uses to separate fields. Supported values are `newline`(\n), `comma`(,), 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`.

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

Note: The `selfDescribing` attribute at this level does not override the message level setting in Artix 1.2.

`<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 four 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.
`bindingOnly`	An optional attribute specifying that the field is only specified in the binding and has no corresponding logical message `part`. Valid settings are `true` and `false`. The default is `false`.
`fixedValue`	An optional attribute specifying the value of a `bindingOnly` field. If `bindingOnly` is set to `false`, this attribute is ignored. If `bindingOnly` is set to `true`, this attribute is required.

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 138.

Example 138:*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 139.

Example 139:*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.
occurs	An optional attribute specifying the number of occurrences of the sequence's child elements in the message. Default is 1.

`alias`	An optional attribute specifying an alias for the sequence that can be used to identify it on the wire.

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 140 shows an Artix contract fragment for such a mapping.

Example 140:*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 140: *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 at 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 attributeb, `name`, that corresponds to the name of the case element in the union's logical description.

`<tagged:case>` must contian 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 141 shows an Artix contract fragment mapping a union to a tagged data format.

Example 141:*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 141:*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>
```

Adding a Tagged Data Binding to an Artix Contract

Overview

Currently Artix does not provide an automated tool to generate tagged data format bindings for logical interfaces defined in an Artix contract. You can either create a new contract for the tagged data binding and operations or hand enter the mapping information.

Using Artix Designer

Currently Artix does not provide an automated tool to generate fixed record length message bindings for logical interfaces defined in an Artix contract. You must hand enter the mapping information or create a new contract in Artix Designer using the fixed record length data description as a starting point.

Using Artix Designer

To create a new contract using fixed record length data complete the following steps:

1. Select **New|Contract From...**.
2. You will see a screen similar to Figure 34.

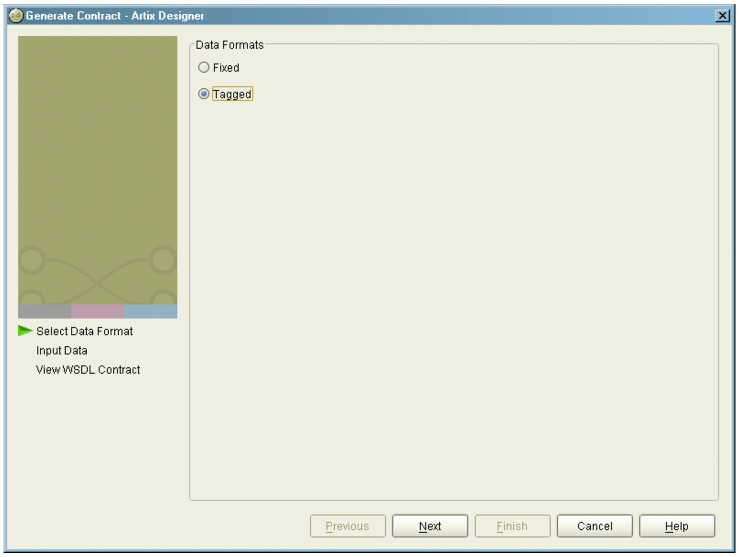
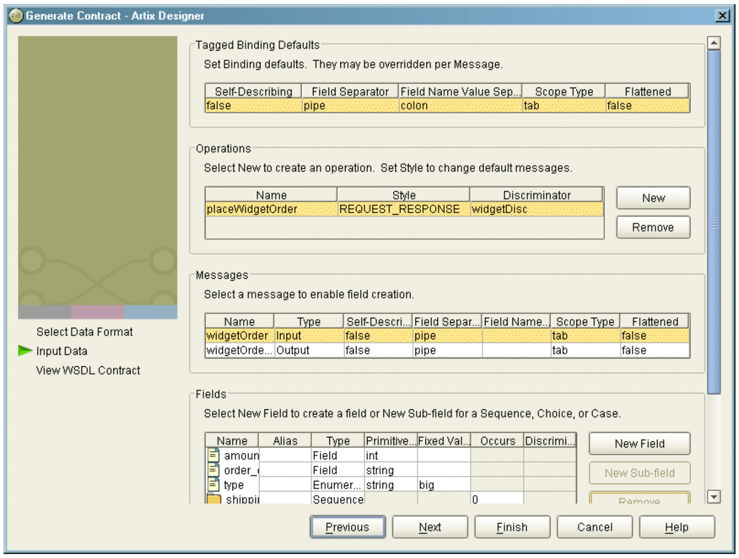


Figure 34: *Binding Selection*

3. Select **Tagged**.
4. Click **Next** to enter the binding information.

5. You will see a screen similar to Figure 35.



Tagged Binding Defaults

Set Binding defaults. They may be overridden per Message.

Self-Describing	Field Separator	Field Name Value Sep...	Scope Type	Flattened
false	pipe	colon	tab	false

Operations

Select New to create an operation. Set Style to change default messages.

Name	Style	Discriminator
placeWidgetOrder	REQUEST_RESPONSE	widgetDisc

Messages

Select a message to enable field creation.

Name	Type	Self-Descri...	Field Separ...	Field Name...	Scope Type	Flattened
widgetOrder	Input	false	pipe		tab	false
widgetOrde...	Output	false	pipe		tab	false

Fields

Select New Field to create a field or New Sub-field for a Sequence, Choice, or Case.

Name	Alias	Type	Primitive...	Fixed Val...	Occurs	Discrimi...
amoun		Field	int			
order_		Field	string			
type		Enumer...	string	big		
shippi		Sequence			0	

Figure 35: *Tagged Binding Information Screen*

6. Under the **Tagged Bindings Defaults** enter the default values for the `selfDescribing`, `fieldSeperator`, `fieldNameSeperator`, `scopeType`, and `flattened` attributes for this binding.

   These attributes of the `<tagged:binding>` tag are described on page 354.

7. Under **Operations** enter the information for the operations your service offers.

8. Under **Messages** enter the messages for the operation selected in the **Operations** field.

   You are able to provide alternate values for the `selfDescribing`, `fieldSeperator`, `fieldNameSeperator`, `scopeType`, and `flattened` attributes here. These values are set on the `<tagged:body>` tag as described on page 355.

9. Under **Fields** enter the fields that make up the message selected in the **Messages** field.

Each message part can be either a field as described in "<tagged:field>" on page 356, an enumeration as described in "<tagged:enumeration>" on page 357, a sequence as described in "<tagged:sequence>" on page 357, or a choice as described in "<tagged:choice>" on page 359.

10. Click **Finish** to create the contract with the tagged data binding.

Example

Example 142 shows an example of an Artix contract containing a tagged data format binding.

Example 142:*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/">
      <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>
```

Example 142:*Tagged Data Format Binding*

```
      <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>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
  </portType>
```

Example 142:*Tagged Data Format Binding*

```
      <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>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
  </portType>
```

Example 142:*Tagged Data Format Binding*

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

Example 142:*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 17

SOAP Payload Format

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 explains how to configure and extend a WSDL contract to use a SOAP binding and a SOAP-over-HTTP port. It provides a description of the WSDL extensions involved in extending a WSDL contract for SOAP. It outlines the XML types supported by SOAP in Artix. Finally, it provides an overview of the WSDL extension schema that supports the use of SOAP with 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?" on page 373.
- "XML" on page 373.
- "XML and Unicode" on page 374.
- "HTTP" on page 374.
- "SOAP specification" on page 375.

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 "Using the HTTP Plug-in" on page 225 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 376 briefly discusses the format and components of SOAP messages, and their use with HTTP. "SOAP Encoding of Data Types" on page 382 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" on page 376.
- "Example of SOAP request message" on page 377.
- "Explanation of SOAP request message" on page 377.
- "Example of SOAP response message" on page 378.
- "Explanation of SOAP response message" on page 379.
- "Example of SOAP response with fault" on page 379.
- "Explanation of SOAP response with fault" on page 380.

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 143:*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 144 shows an example of the format of a typical SOAP request message, based on Example 143 on page 376 (in this case, the client has passed student code `815637` and subject `History` as input parameters):

Example 144:*Example of a SOAP Request Message*

```
1  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'?>
2  <SOAP-ENV:Envelope
       xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
       SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/"/>
3          <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 144 on page 377 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 "Using the HTTP Plug-in" on page 225 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 382 for more details of these).

   In Example 144 on page 377, 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 145 shows an example of the format of a typical SOAP response message, based on Example 143 on page 376 (in this case, the server has returned grade `A`):

Example 145:*Example of a SOAP Response Message*

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

   <?xml version="1.0" encoding='UTF-8'?>
2  <SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/"/>
3          <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 145 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 "Using the HTTP Plug-in" on page 225 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 377.
3. The explanation of the SOAP Body element is the same as in "Explanation of SOAP request message" on page 377, 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 146 shows an example of the format of a typical SOAP response message indicating an error.

Example 146:*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 146:*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 146 on page 379 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 "Using the HTTP Plug-in" on page 225 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 146 on page 379 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?" on page 382.
- "Role of SOAP encoding" on page 382.
- "SOAP encoding styles" on page 384.
- "Encoding simple types" on page 384.
- "Encoding complex struct types" on page 386.
- "Encoding complex array types" on page 388.

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 373, 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 36 on page 383 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 36, all data transferred as part of a SOAP payload is marshalled across the network as UTF-encoded binary strings.

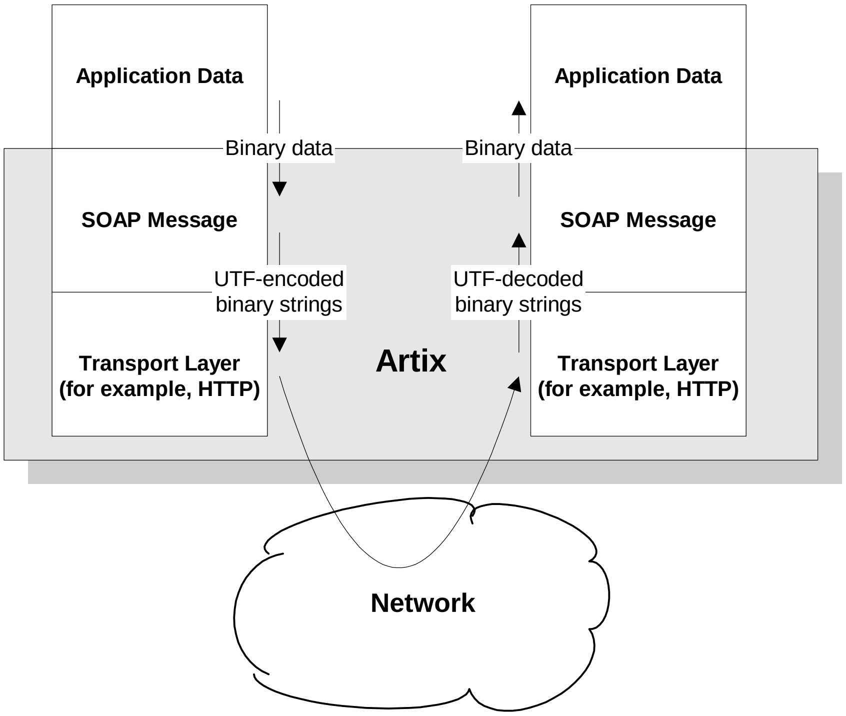

Application Data
Application Data
Binary data
Binary data
SOAP Message
SOAP Message
UTF-encoded binary strings
UTF-decoded binary strings
Transport Layer (for example, HTTP)
Artix
Transport Layer (for example, HTTP)
Network

Figure 36: *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 388 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 384, 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.

Adding a SOAP Binding

Overview

You can configure an Artix WSDL contract with various extensions that support the use of a SOAP binding with Artix. This section describes how to use the **Artix Designer** GUI to add a SOAP binding to a WSDL contract. It discusses the following topics:

- "GUI steps" on page 390.
- "WSDL example" on page 395.

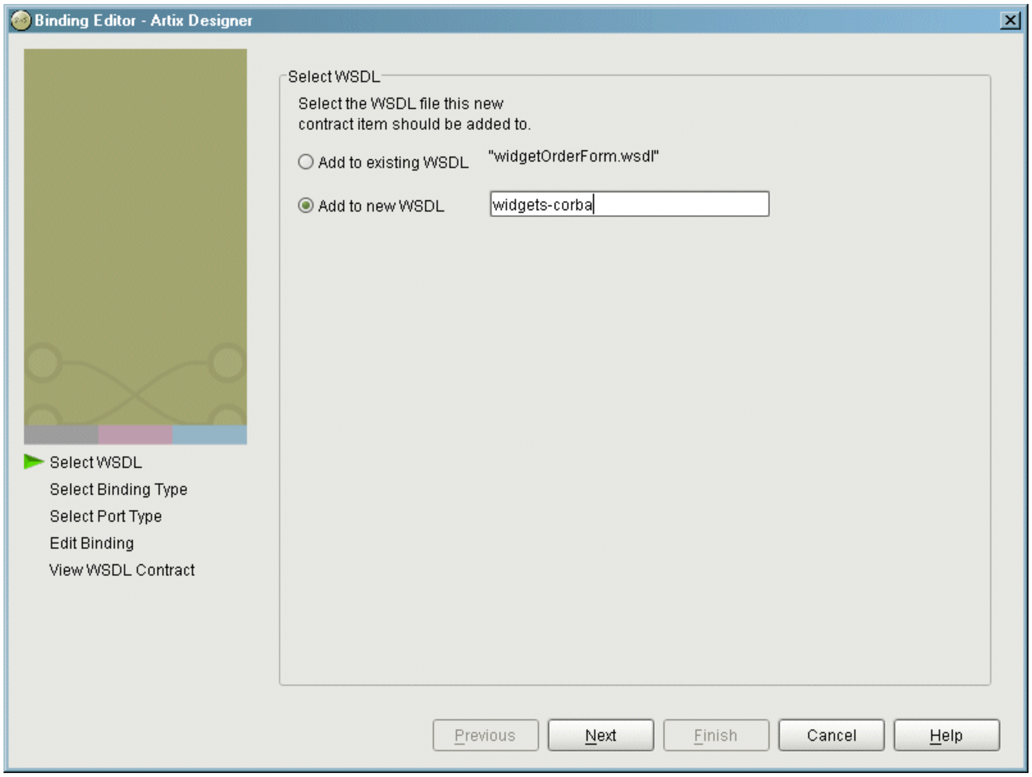
Note: This section deals specifically with how to set up a SOAP binding within an Artix WSDL contract. It assumes that you have already set up the logical components of the contract relating to types, messages, and port types.

GUI steps

To add a SOAP binding to your service contract, using the **Artix Designer** GUI, complete the following steps:

1. From the project tree, select the contract to which you want to add the SOAP binding.
2. Select **New|Binding** from the **Contract** menu of the designer.

3. You will see a screen like Figure 37.



Select WSDL

Select the WSDL file this new contract item should be added to.

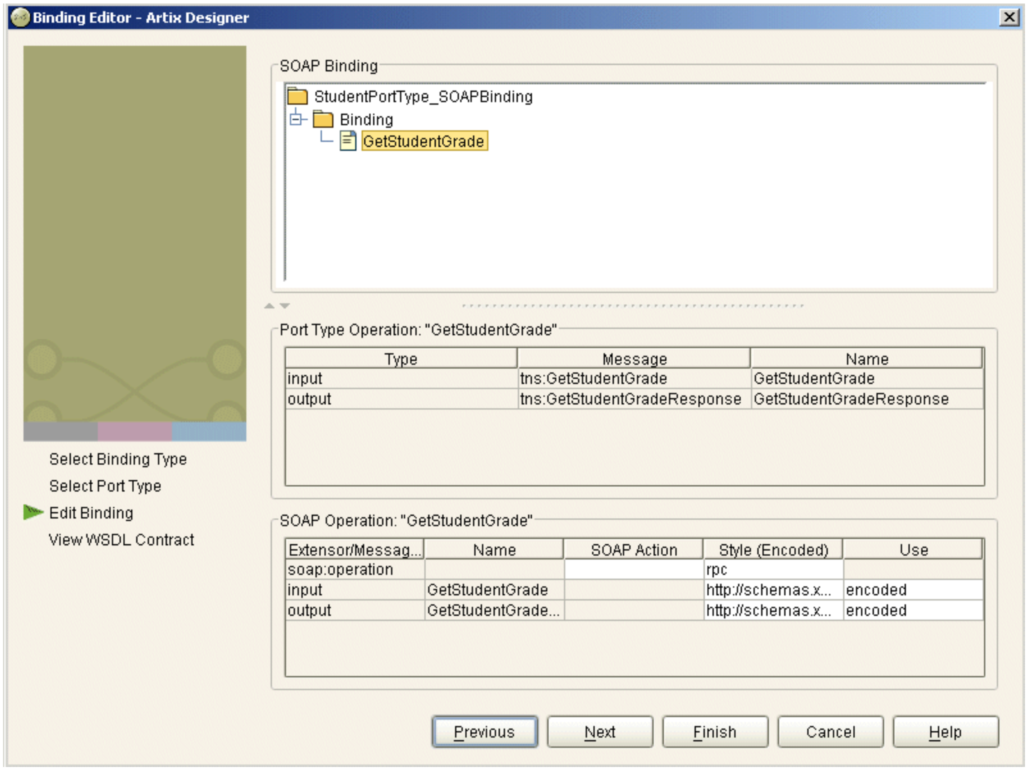
( ) Add to existing WSDL "widgetOrderForm.wsdl"

(●) Add to new WSDL widgets-corba

Figure 37: *Select WSDL location*

4. Select where to create the WSDL entry for the new binding.
   - **Add to existing WSDL** adds the routing information to the bottom of the existing contract and does not make a back-up of the non-routed WSDL file.
   - **Add to new WSDL** creates a new WSDL document that contains the routing information and imports the original WSDL document.
5. Click **Next**.
6. Select SOAP as your binding type.

7. Click **Next**.
8. From the **Port Type** drop down list, select the port type that the binding relates to.
9. Type a name for your binding in the **Binding Name** field, or accept the default that consists of the port type name with a _SOAPBinding suffix.
10. From the **Use** drop down list, select either **encoded** or **literal**, to indicate whether message parts are to consist of abstract type definitions or concrete schema definitions. The value you choose is subsequently populated in the `soap:body` use attribute in your WSDL contract. See "soap:body element" on page 410 for more details.
11. From the **Style** drop down list, select either **rpc** or **document**, to indicate whether message parts pertaining to each operation are to consist of RPC-based parameters and return values or document-based body entries by default. The value you choose is subsequently populated in the `soap:binding style` attribute in your WSDL contract. See "soap:binding element" on page 407 for more details.
12. Click **Next**.
13. Click on the name of an operation within your binding. The screen then appears as shown in Figure 38. (For this example, assume that **encoded** was selected in point 7, and **rpc** was selected in point 8.)



SOAP Binding

- StudentPortType_SOAPBinding
  - Binding
    - GetStudentGrade

Port Type Operation: "GetStudentGrade"

Type	Message	Name
input	tns:GetStudentGrade	GetStudentGrade
output	tns:GetStudentGradeResponse	GetStudentGradeResponse

SOAP Operation: "GetStudentGrade"

Extensor/Messag...	Name	SOAP Action	Style (Encoded)	Use
soap:operation			rpc	
input	GetStudentGrade		http://schemas.x...	encoded
output	GetStudentGrade...		http://schemas.x...	encoded

Figure 38: *Editing a SOAP Binding for an Operation*

14. If you want to include a SOAPAction field in the HTTP header of a SOAP message, use the highlighted **SOAP Action** field to type the URL that represents the resource being requested by the operation.

    Note: This step only relates to the use of SOAP over HTTP, but it is not mandatory for the purposes of Artix. It is available in case some third-party SOAP servers that do use a SOAPAction field in their HTTP headers are to be contacted.

15. If you want to override for a particular operation the default setting for **Style** that you set in point 8, delete the default value and type the new value in the relevant field in the **Style (Encoded)** column. (See point 8 for more details of valid values.)
16. If you want to override for a particular operation the default setting for **Use** that you set in point 7, delete the default value and type the new value in the relevant field(s) in the **Use** column. (See point 7 for more details of valid values.)
17. If you want to use one or more customized encoding styles, add the URL(s) relating to each customized encoding style to the relevant field(s) in the Style (Encoded) column.

    Note: If you want this field to contain more than one URL, ensure that each URL is separated by a space, and the URLs are ordered according to the most restrictive set of rules first and least restrictive set of rules last.

18. Click **Next**.
19. Review the WSDL for the new SOAP binding. See Example 147 for an example of how the WSDL might appear.
20. If it is correct, click **Finish**.

WSDL example

Example 147 provides an example of how the WSDL might appear for a SOAP binding in an Artix contract.

Example 147:*Example of WSDL for a SOAP Binding*

```
<wsdl:binding name="StudentPortType_SOAPBinding" type="ns1:StudentPortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetStudentGrade">
        <soap:operation soapAction="" style="rpc"/>
        <wsdl:input name="GetStudentGrade">
            <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       use="encoded"/>
        </wsdl:input>
        <wsdl:output name="GetStudentGradeResponse">
            <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       use="encoded"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

Adding a Port for SOAP over HTTP

Overview

You can configure an Artix WSDL contract with various extensions that support the use of a SOAP-over-HTTP port. This section describes how to use the **Artix Designer** GUI to add to a WSDL contract a port that enables the use of SOAP over HTTP.

Note: This section is only relevant if you want to use SOAP over HTTP. If you are using SOAP with another transport type, ignore this section and see the relevant chapter in this guide that pertains to that transport type.

In this section

This section discusses the following topics:

Adding a Port for Non-Secure Connections

Overview

This subsection describes how to use the **Artix Designer** GUI to add a port for SOAP over HTTP that does not enable secure connections. It discusses the following topics:

- "GUI steps" on page 397.
- "WSDL example" on page 400.

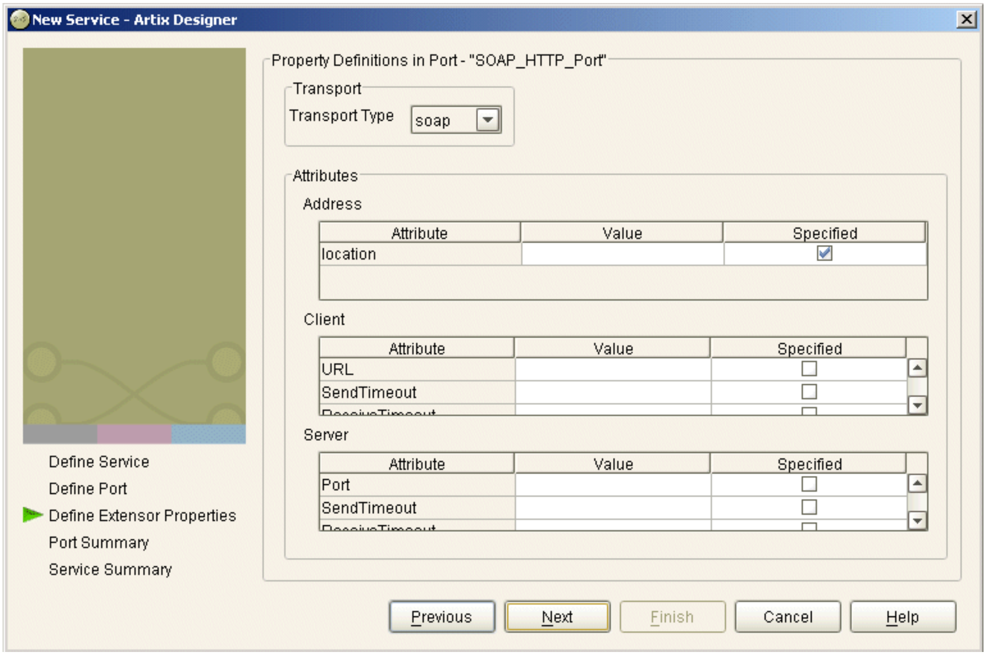
Note: This section deals specifically with how to set up port information within the `<service>` component of a WSDL contract. To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See "Adding a SOAP Binding" on page 390 or the relevant chapter relating to the payload format you are using for more details about setting up a binding in a WSDL contract.

GUI steps

Complete the following steps to add a port to your service contract, using the **Artix Designer** GUI, to enable the use of SOAP over HTTP:

1. From the project tree, select the contract to which you want to add the port.
2. Select **Services|New Service** from the **Contract** menu of the designer.
3. Enter a unique name for the new service.
4. Click **Next**.
5. Enter a name for the new port that is being created.
6. From the **Binding** drop down list, select the binding that the port is going to expose.

7. Click **Next** to open the screen shown in Figure 39.



Property Definitions in Port - "SOAP_HTTP_Port"

Transport

Transport Type: soap

Attributes

Address

Attribute	Value	Specified
location		✓

Client

Attribute	Value	Specified
URL		☐
SendTimeout		☐

Server

Attribute	Value	Specified
Port		☐
SendTimeout		☐

Figure 39: *Selecting a SOAP Transport Type*

8. Ensure that SOAP is selected as your transport type.
9. In the **Value** field corresponding to the **location** line of the **Address** configuration table, type the URL that represents the resource being requested.

Note: The **Address** configuration table relates to the `soap:address` element within the port component of the WSDL contract. You must specify a value for the **location** attribute. See “SOAP WSDL Extensions” on page 405 for more details of the `soap:address location` attribute.

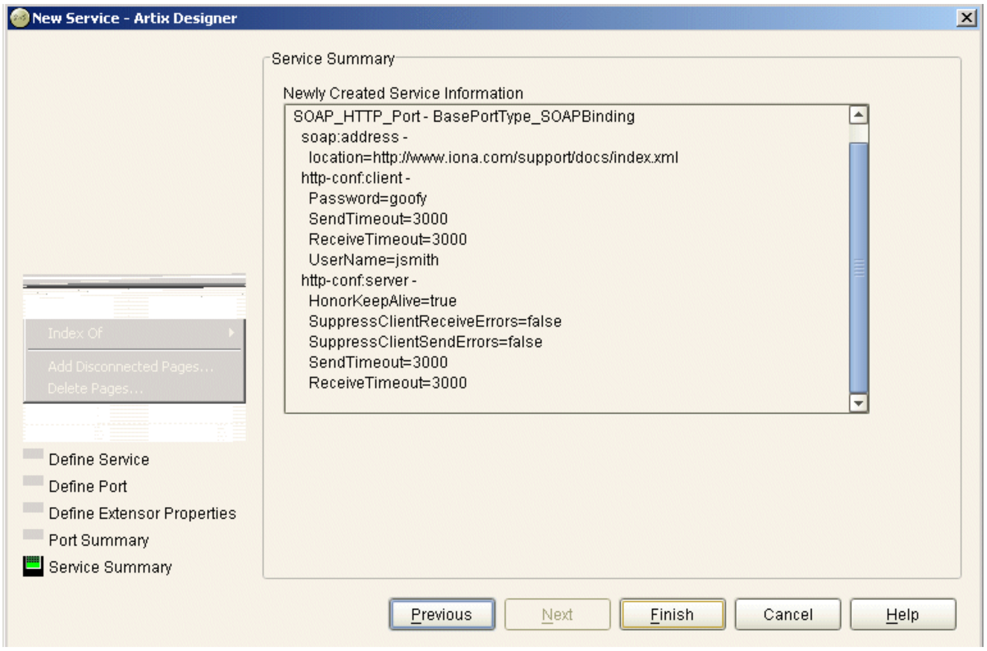
10. To specify a value for another attribute, place a check in the **Specified** box on the appropriate line in the appropriate configuration table, and type or (in the case of certain true or false attributes) select the value you want.

    Note: All attributes are optional in the **Client** and **Server** configuration tables. These relate to the `http-conf:client` and `http-conf:server` elements that can be specified as peers of the `soap:address` element under the same port binding. See "Using the HTTP Plug-in" on page 225 for details of each attribute relating to `http-conf:client` and `http-conf:server`.

11. Click **Next**.
12. Review the settings for the new port.
13. If it is correct, click **Next**.
14. Review the settings for the new service in which the port is described.
15. If it is correct, click **Finish**.

WSDL example

Figure 40 shows an example summary of port configuration settings in the GUI.



```
SOAP_HTTP_Port - BasePortType_SOAPBinding
 soap:address -
  location=http://www.iona.com/support/docs/index.xml
 http-conf:client -
  Password=goofy
  SendTimeout=3000
  ReceiveTimeout=3000
  UserName=jsmith
 http-conf:server -
  HonorKeepAlive=true
  SuppressClientReceiveErrors=false
  SuppressClientSendErrors=false
  SendTimeout=3000
  ReceiveTimeout=3000
```

Figure 40: *Example Set of SOAP_HTTP Configuration Settings in GUI*

Example 148 shows the WSDL extract that is subsequently generated for the service component of your Artix contract, based on the example settings in Figure 40. As shown in Example 148, client and server HTTP configuration attributes are contained respectively within elements, called `http-conf:client` and `http-conf:server`, which are peers of the `soap:address` element.

Example 148:*Extract of Example WSDL Contract*

```
<wsdl:service name="BaseService">
    <wsdl:port binding="ns1:StudentPortType_SOAPBinding" name="SOAP_HTTP_Port">
        <soap:address location="http://www.iona.com/support/docs/index.xml"/>
        <http-conf:client Password="goofy" ReceiveTimeout="3000" SendTimeout="3000"
                          UserName="jsmith"/>
        <http-conf:server HonorKeepAlive="true" ReceiveTimeout="3000"
                          SendTimeout="3000" SuppressClientReceiveErrors="false"
                          SuppressClientSendErrors="false"/>
    </wsdl:port>
</wsdl:service>
```

Adding a Port for Secure Connections

Overview

This subsection describes how to use the **Artix Designer** GUI to add a port for SOAP over HTTP that enables secure connections. It discusses the following topics:

- "SSL-related attributes" on page 402.
- "GUI steps" on page 403.
- "WSDL example" on page 403.

Note: This section deals specifically with how to set up port information within the `<service>` component of a WSDL contract. To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See the chapter relating to the payload format you are using for more details about setting up a binding for it in a WSDL contract.

SSL-related attributes

The SSL-related attributes that can be configured to be included in the `<http-conf:client>` and `<http-conf:server>` elements of an HTTP port binding are as follows:

Client SSL Attributes	Server SSL Attributes
`UseSecureSockets`	`UseSecureSockets`
`ClientCertificate`	`ServerCertificate`
`ClientCertificateChain`	`ServerCertificateChain`
`ClientPrivateKey`	`ServerPrivateKey`
`ClientPrivateKeyPassword`	`ServerPrivateKeyPassword`
`TrustedRootCertificate`	`TrustedRootCertificate`

See "Using the HTTP Plug-in" on page 225 more details of these attributes.

GUI steps

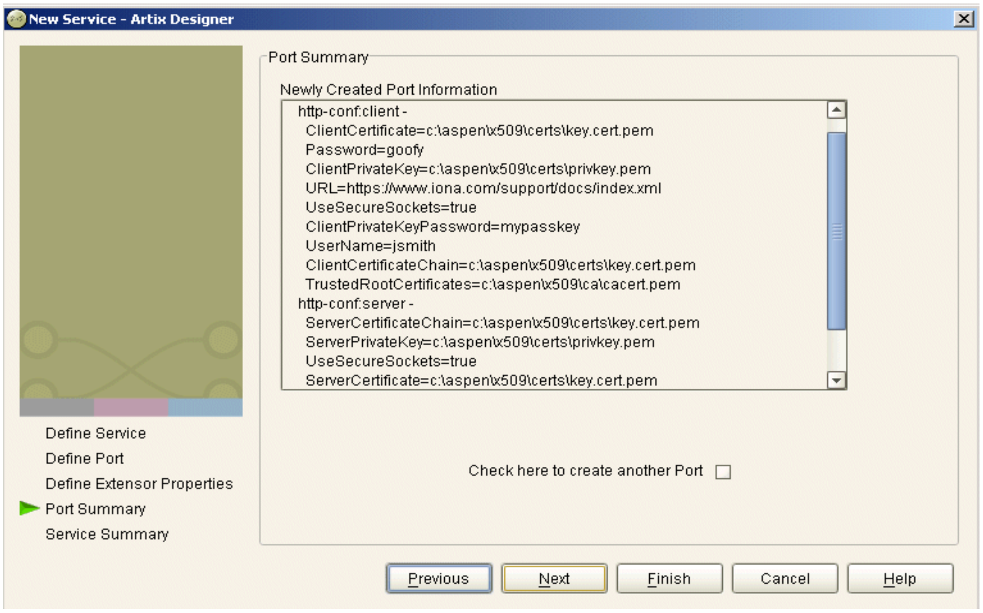
All the GUI steps described in "GUI steps" on page 397 are relevant and should be followed here, with the following stipulations:

- Specify `https://` rather than `http://` as the prefix for the value of the **location** attribute in the **Address** configuration table.
- Enter values for the various SSL-related attributes in the **Client** and **Server** configuration tables. See "SSL-related attributes" on page 402 for a listing of these attributes.

Note: When you specify `https://` as the prefix for the value of the **location** attribute in the **Address** configuration table, a secure HTTP connection is automatically enabled, even if **UseSecureSockets** is not set to `true`.

WSDL example

Figure 41 shows an example summary of SSL-related HTTP configuration settings in the GUI


New Service - Artix Designer
Port Summary
Newly Created Port Information
Define Service
Define Port
Define Extensor Properties
Port Summary
Service Summary
Check here to create another Port
Previous
Next
Finish
Cancel
Help

```
http-conf:client -
  ClientCertificate=c:\aspen\x509\certs\key.cert.pem
  Password=goofy
  ClientPrivateKey=c:\aspen\x509\certs\privkey.pem
  URL=https://www.iona.com/support/docs/index.xml
  UseSecureSockets=true
  ClientPrivateKeyPassword=mypasskey
  UserName=jsmith
  ClientCertificateChain=c:\aspen\x509\certs\key.cert.pem
  TrustedRootCertificates=c:\aspen\x509\ca\cacert.pem
http-conf:server -
  ServerCertificateChain=c:\aspen\x509\certs\key.cert.pem
  ServerPrivateKey=c:\aspen\x509\certs\privkey.pem
  UseSecureSockets=true
  ServerCertificate=c:\aspen\x509\certs\key.cert.pem
```

Figure 41: *Example Set of SSL-Related HTTP Configuration Settings*

Example 149 shows the WSDL extract that is subsequently generated for the service component of your Artix contract, based on the example settings in Figure 41 on page 403. As shown in Example 149, client and server HTTP configuration attributes are contained respectively within elements called `http-conf:client` and `http-conf:server`.

Example 149:*Extract of Example WSDL Contract with SSL Attributes*

```
<wsdl:service name="BaseService">
    <wsdl:port binding="ns1:StudentPortType_SOAPBinding" name="SOAP_HTTP_Port">
        <soap:address location="http://www.iona.com/support/docs/index.xml"/>
        <http-conf:client ClientCertificate="c:\aspen\x509\certs\key.cert.pem"
                          ClientCertificateChain="c:\aspen\x509\certs\key.cert.pem"
                          ClientPrivateKey="c:\aspen\x509\certs\privkey.pem"
                          ClientPrivateKeyPassword="mykeypass" Password="goofy"
                          TrustedRootCertificates="c:\aspen\x509\ca\cacert.pem"
                          UseSecureSockets="true"
                          Password="goofy"
                          UserName="jsmith"/>
        <http-conf:server ServerCertificate="c:\aspen\x509\certs\key.cert.pem"
                          ServerCertificateChain="c:\aspen\x509\certs\key.cert.pem"
                          ServerPrivateKey="c:\aspen\x509\certs\privkey.pem"
                          ServerPrivateKeyPassword="mykeypass"
                          TrustedRootCertificates="c:\aspen\x509\ca\cacert.pem"
                          UseSecureSockets="true"/>
    </wsdl:port>
</wsdl:service>
```

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:

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 150 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 150:*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" on page 407.
- "soap:operation element" on page 409.
- "soap:body element" on page 410.
- "soap:fault element" on page 412.
- "soap:address element" on page 413.

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 33 describes the attributes defined within the soap:binding element.

Table 33: *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 33: *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 143 on page 376) 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 143 on page 376) 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 33 describes the attributes defined within a `soap:operation` element.

Table 34: *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 407 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 409, the structure of the SOAP Body within a SOAP message is dependent on the setting of the soap:operation style attribute.

Table 33 describes the attributes defined within the soap:body element.

Table 35: *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 35: *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 35: *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 33 describes the attributes defined within the `soap:fault` element.

Table 36: *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 410 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 411 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 33 describes the `location` attribute defined within the `soap:address` element.

Table 37: *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`
	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 415.
- "Other supported types" on page 416.

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>`

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.

I

IntelliJ

An integrated development environment provided by JetBrains. The Artix developer tools are accessed via this environment.

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 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.

System Designer

The main design tool within the Artix development tool suite. This component lets the developer graphically describe the integration project in terms of hosts, systems, services, and connections.

System Diagram

A diagram produced by System Designer, which represents the integration project being solved by Artix.

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