Artix™

Developing Artix Applications in Java

Version 2.1, July 2004

Making Software Work Together™

Updated: 15-Nov-2004

M 3 2 1 0

Contents

List of Figures v

List of Tables vii

Preface ix
- What is Covered in this Book ix
- Who Should Read this Book ix
- How to Use this Book ix
- Online Help x
- Finding Your Way Around the Artix Library xi
- Additional Resources for Information xii
- Typographical Conventions xii
- Keying conventions xiii

Chapter 1 Understanding the Artix Java Development Model 1
- Separating Transport Details from Application Logic 2
- Representing Services in Artix Contracts 4
- Mapping from an Artix Contract to Java 6

Chapter 2 Developing Artix Enabled Clients and Servers 9
- Generating Stub and Skeleton Code 10
- Java Package Names 13
- Developing a Server 15
- Developing a Client 20
- Building an Artix Application 24

Chapter 3 Advanced Programming Issues 25
- Servant Registration 26
  - Static Servant Registration 27
  - Transient Servant Registration 28
- Proxy Creation 30
- Getting a Bus 32

Threading 33
Setting Client Connection Attributes Using the Stub Interface 37
Class Loading 41

Chapter 4 Working with Artix Data Types 45
Using Native XMLSchema Simple Types 47
Simple Type Mapping 48
Special Simple Type Mappings 50
Unsupported Simple Types 52
Defining Your Own Simple Types 53
Using XMLSchema Complex Types 56
Sequence and All Complex Types 57
Choice Complex Types 64
Attributes 68
Nesting Complex Types 72
Deriving a Complex Type from a Simple Type 78
Occurrence Constraints 81
Using XMLSchema any Elements 84
SOAP Arrays 92
Lists 95
Enumerations 98
Deriving Types Using <complexContent> 104
Holder Classes 107
Using SOAP with Attachments 111

Chapter 5 Creating User-Defined Exceptions 117
Describing User-defined Exceptions in an Artix Contract 118
How Artix Generates Java User-defined Exceptions 120
Working with User-defined Exceptions in Artix Applications 122

Chapter 6 Working with Artix Type Factories 125
Introduction to Type Factories 126
Registering Type Factories 128
Getting Type Information From Type Factories 131

Chapter 7 Working with XMLSchema anyTypes 135
Introduction to Working with XMLSchema anyTypes 136
Setting anyType Values 138

Retrieving Data from anyTypes **140**

Chapter 8 Artix References **145**
Introduction to Working with References **146**
Reference Basic Concepts 147
Creating References 151
Instantiating Service Proxies Using a Reference 153
Using References in a Factory Pattern **154**
Bank Service Contract 155
Bank Service Implementation 162
Bank Service Client 165
Using References to Implement Callbacks **168**
The Accounting Contract 169
The Accounting Client 175
The Accounting Server 180

Chapter 9 The Artix Locator **183**
Overview of the Locator **184**
Locator WSDL **187**
Registering Endpoints with the Locator **191**
Reading a Reference from the Locator **192**

Chapter 10 Using Message Contexts **197**
Understanding Message Contexts in Artix **198**
Getting the Context Registry 201
Getting the Message Context for a Thread 203
Working with Generic Contexts 206
Working with Artix Message Contexts 211
Sending Header Information Using Contexts **218**
Defining Context Data Types 219
Registering Context Types 221
SOAP Header Example 223

Chapter 11 Developing Java Plug-Ins **233**
Extending the BusPlugIn Class **234**
Implementing the BusPlugInFactory Interface **237**

Chapter 12 Writing Message Handlers 239
Message Handlers: An Introduction 240
Developing Request-Level Handlers 244
Developing Message-Level Handlers 251

Chapter 13 Artix IDL to Java Mapping 259
Introduction to IDL Mapping 260
IDL Basic Type Mapping 262
IDL Complex Type Mapping 264
IDL Module and Interface Mapping 277

Glossary 281

Index 291

List of Figures

Figure 1: SingleInstanceServant 34
Figure 2: SerializedServant 35
Figure 3: PerInvocationServant 36
Figure 4: Class Loader Firewall 41
Figure 5: Artix Locator Overview 184
Figure 6: Steps to Read a Reference from the Locator 192
Figure 7: Overview of the Message Context Mechanism 199
Figure 8: Contexts Passed Along Request/Reply Chain 211
Figure 9: The Life of a Message 240
Figure 10: Handler Levels 241
Figure 11: Artix and CORBA Alternatives for IDL to Java Mapping 261

List of Tables

Table 1: discover-source values for the Class Loader Firewall 42
Table 2: Simple Schema Type to Primitive Java Type Mapping 48
Table 3: simple Schema Type to Java Wrapper Class Mapping 51
Table 4: Attributes for an any 84
Table 5: List Type Facets 95
Table 6: MIME Type Mappings 111
Table 7: anyType Setter Methods for Primitive Types 138
Table 8: Methods for Extracting Primitives from AnyType 141
Table 9: Artix Mapping of IDL Basic Types to Java 262

Preface

What is Covered in this Book

Developing Artix Applications in Java discusses the main aspects of developing transport-independent services and service consumers using Java stub and Java skeleton code generated by Artix. This book covers:

- how to access the Artix bus
- how to use generated data types
- how to create user defined exceptions
- how to access the header information for the transports supported by Artix.

Who Should Read this Book

Developing Artix Applications in Java is intended for Artix Java programmers. In addition to a knowledge of Java, this guide assumes that the reader is familiar with the basics of WSDL and XML schemas. Some knowledge of Artix concepts would be helpful, but is not required.

How to Use this Book

If you are new to using Artix to develop Java applications, Chapter 1 provides an overview of the benefits of using Artix and how Artix generates Java code from an Artix contract.

If you are interested in the basics of writing an Artix-enabled service or service consumer, Chapter 2 describes the basic steps to implement a service, connect to the Artix bus, and create JAX-RPC compliant proxies using Artix-generated code.

Chapter 3 extends the discussion of building Artix applications. It includes details about the threading model used by Java Artix applications, using Artix specific methods for creating proxies, and class loading issues that may be encountered when using Artix.

If you need help understanding how to work with the classes generated to represent complex data types, Chapter 4 gives detailed description of how all of the XMLSchema data types in an Artix contract are mapped into Java code. It also contains details and examples on using the generated Java code.

If you want to create user-defined exceptions, Chapter 5 explains how to describe a user-defined exception in an Artix contract and how exceptions are mapped into Java code by Artix.

If you want to learn how to develop Java code to use XMLSchema anyType elements, Chapter 7 describes how they are mapped into Java and describes the Artix classes that allow you to work with them.

Chapter 8 describes how Artix references work. It describes the basic concepts and APIs for working with references in Artix. In addition, it gives detailed examples of using references in a factory pattern and in developing callbacks.

If you want to use SOAP headers, Chapter 10 describes how to use Artix message contexts to set data into a SOAP header.

If you want to write message handlers for doing advanced message processesing, Chapter 12 describes how to develop and configure message handlers for use in Artix applications.

Online Help

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

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

There are two ways that you can access the Online Help:

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

Finding Your Way Around the Artix Library

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

If you are new to Artix

You may be interested in reading *Learning about Artix.* It describes basic Artix concepts and guides you through a number of Artix programming examples.

To design Artix solutions

You should read *Designing Artix Solutions*. It provides detailed information about creating WSDL-based Artix contracts, Artix stub and skeleton code, and the artifacts needed to deploy Artix solutions.

To develop applications using Artix stub and skeleton code

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

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

To manage and configure your Artix solution

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

If you want to know more about Artix security

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

Have you got the latest version?

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs. Compare the version details provided there with the last updated date printed on the inside cover of the book you are using (at the bottom of the copyright notice).

Additional Resources for Information

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 doc-feedback@iona.com.

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

http://www.iona.com/support/knowledge_base/index.xml

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

http://www.iona.com/support/update/

Typographical Conventions

This book uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 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

Understanding the Artix Java Development Model

The Artix Java development tools generate JAX-RPC compliant Java code from WSDL-based Artix contracts. Using the generated code, you can develop transport-independent applications that take advantage of the Artix bus.

In this chapter

This chapter discusses the following topics:

Separating Transport Details from Application Logic	page 2
Representing Services in Artix Contracts	page 4
Mapping from an Artix Contract to Java	page 6

Separating Transport Details from Application Logic

Overview

One of the main benefits of using Artix to develop applications is that it removes the network protocol details, message transport details, and payload format details from the business of developing application logic. Artix enables developers to write robust applications using standard Java APIs and leaves the nitty-gritty of the messaging mechanics up to the system administrators or system architects.

Unlike CORBA or J2EE, however, Artix does not provide this abstraction from the transport details by limiting the types of messaging system the application can work on. It makes the application capable of using any number of transports and payload formats. In addition, Artix allows applications in the same system to interoperate across multiple messaging protocols.

Dividing the logical and physical

Artix achieves this separation of the logical part of an application from the physical details of how data is passed by describing applications using Web Services Description Language (WSDL) as the basis for Artix contracts. Artix contracts are XML documents that describe applications in two sections:

Logical:

The logical section of an Artix contract defines the abstract data types used by the application, the logical operations exposed by the application, and the messages passed by those operations.

Physical:

The physical section of an Artix contract defines how the messages used by the application are mapped for transport across the network and how the application's port is configured. For example, the physical section of the contract would be where it is made explicit that an application will use SOAP over HTTP to expose its operations.

The Artix bus

The Artix bus is a library that provides the layer of abstraction to liberate the application logic from the transport once the code is generated. The bus reads the transport details from the physical section of the Artix contract, loads the appropriate payload and transport plug-ins, and handles the mapping of the data onto and off the wire.

The bus also provides access to the message headers so you can add payload-specific information to the data if you wish. In addition, it provides access to the transport details to allow dynamic configuration of transports.

Representing Services in Artix Contracts

Overview

Services, which are the operations exposed by an application, are described in the logical section of an Artix contract. When defining a service in an Artix contract, you break it down into three parts: the complex data types used in the messages, the messages used by the operations, and the collection of operations that make up the service.

Data types

Complex data types, such as arrays, structures, and enumerations, are described in an Artix contract using XMLSchema. The descriptions are contained within the WSDL `<types>` element. The data type descriptions represent the logical structure of the data. For example, an array of integers could be described as shown in Example 1.

Example 1: *Array Description*

```
<complexType name="ArrayOfInt">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="item"
             type="xsd:int"/>
  </sequence>
</complexType>
```

The described types are used to define the message parts used by the service.

Messages

In an Artix contract messages represent the data passed to and received from a remote system in the execution of an operation. Messages are described using the `<message>` element and consist of one or more `<part>` elements. Each message part represents an argument in an operation's parameter list or a piece of data returned as part of an exception.

Service

In an Artix contract logical services are described using the `<portType>` element and consist of one or more `<operation>` elements. Each `<operation>` element describes an operation that is to be exposed over the network.

Operations are defined by the messages which are passed to and from the remote system when the operation is invoked. In an Artix contract, each operation is allowed to have one input message, one output message, and any number of fault messages. It does not need to have any of these elements. An input message describes the parameter list passed into the operation. An output message describes the return value, and the output parameters of the operation. A fault message describes an exception that the operation can throw. For example, a Java method with the signature `long myOp(char c1, char c2)`, would be described as shown in Example 2.

Example 2: *Operation Description*

```
<message name="inMessage">
  <part name="c1" type="xsd:char" />
  <part name="c2" type="xsd:char" />
</message>
<message name="outMessage">
  <part name="returnVal" type="xsd:int" />
</message>
<portType name="myService">
  <operation name="myOp">
    <input message="inMessage" name="in" />
    <output message="outMessage" name="out" />
  </operation>
</portType>
```

Mapping from an Artix Contract to Java

Overview

Artix maps the WSDL-based Artix contract description of a service into Java server skeletons and client stubs following the JAX-RPC specification. This allows application developers to implement the service's logic using standard Java and be assured that the service will be interoperable with a wide range of other services.

Ports

For each `<port>` element in an Artix contract, a Java interface that extends `java.rmi.Remote` is generated. The name of the generated interface is taken from the `name` attribute of the `<port>` element. The interface's name will be identical to the `<port>` elements's name unless the `<port>` element's name ends in `Port`. In this case, the `Port` will be stripped off the interface's name.

The generated interface will contain each of the operations of the `<portType>` to which the `<port>` element is bound. For example, the contract shown in Example 3 will generate an interface, `sportsCenter`, containing one operation, `update`.

Example 3: *SportsCenter Port*

```
<message name="scoreRequest">
  <part name="teamName" type="xsd:string" />
</message>
<message name="scoreReply">
  <part name="score" type="xsd:int" />
</message>
<portType name="sportsCenterPortType">
  <operation name="update">
    <input message="scoreRequest" name="request" />
    <ouput message="scoreReply" name="reply" />
  </operation>
</portType>
<binding name="scoreBinding" type="tns:sportsCenterPortType">
...
<service name="sportsService">
  <port name="sportsCenterPort" binding="tns:scoreBinding">
...
```

The generated Java interface is shown in Example 4.

Example 4: *SportsCenter Interface*

```
//Java
public interface sportsCenter extends java.rmi.Remote
{
  int update(String teamName)
    throws java.rmi.RemoteException;
}
```

Operations

Every <operation> element in a contract generates a Java method within the interface defined for the <operation> element's <portType>. The generated method's name is taken from the <operation> element's name attribute. <operation> elements with the same name attribute will generate overloaded Java methods in the interface.

All generated Java methods throw a java.rmi.RemoteException exception. In addition, all <fault> elements listed as part of the operation create an exception to the generated Java method.

Message parts

The message parts of the operation's <input> and <output> elements are mapped as parameters in the generated method's signature. The order of the mapped parameters can be specified using the <operation> element's parameterOrder attribute. If this attribute is used, it must list all of the parts of the input message. The message parts listed in the parameterOrder attribute will be placed in the generated method's signature in the order specified. Unlisted message parts will be placed in the method signature according to the order the parts are specified in the <message> elements of the contract. The first unlisted output message part is mapped to the generated method's return type. The parameter names are taken from the <part> element's name attribute. If the parameterOrder attribute is not specified, input message parts are listed before output message parts. Message parts that are listed in both the input and output messages are considered inout parameters and are listed only according to their position in the input message.

All inout and output message parts, except the part mapped to the return value of the generated method, are passed using Java Holder classes. For the XML primitive types, the Java Holder class used is the standard Java Holder class, defined in javax.xml.rpc.holders package, for the

appropriate Java type. For complex types defined in the contract, the code generator will generate the appropriate `Holder` classes. For more information on data type mapping, see "Working with Artix Data Types" on page 45.

For example, the contract fragment shown in Example 5 would result in an operation, `final`, with a return type of `String` and a parameter list that contains two input parameters and three output parameters.

Example 5: *SportsFinal Port*

```
<message name="scoreRequest">
  <part name="team1" type="xsd:string" />
  <part name="team2" type="xsd:string" />
</message>
<message name="scoreReply">
  <part name="winTeam" type="xsd:string" />
  <part name="team1score" type="xsd:int" />
  <part name="team2score" type="xsd:int" />
</message>
<portType name="sportsFinalPortType">
  <operation name="final">
    <input message="scoreRequest" name="request" />
    <ouput message="scoreReply" name="reply" />
  </operation>
</portType>
<binding name="scoreBinding" type="tns:sportsFinalPortType">
...
<service name="sportsService">
  <port name="sportsFinalPort" binding="tns:scoreBinding">
...
```

The generated Java interface is shown in Example 6.

Example 6: *SportsFinal Interface*

```
//Java
public interface sportsFinal extends java.rmi.Remote
{
  String final(String team1, String team2,
               IntHolder team1score, IntHolder team2score)
    throws java.rmi.RemoteException;
}
```

CHAPTER 2

Developing Artix Enabled Clients and Servers

Artix generates stub and skeleton code that provides a developer with a simple model to develop transport-independent applications.

In this chapter

This chapter discusses the following topics:

Generating Stub and Skeleton Code	page 10
Java Package Names	page 13
Developing a Server	page 15
Developing a Client	page 20
Building an Artix Application	page 24

Generating Stub and Skeleton Code

Overview

The Artix development tools include a utility to generate server skeleton and client stub code from an Artix contract. The generated code is similar to code generated by a CORBA IDL compiler. There are two major differences between CORBA-generated code and Artix-generated code:

- Artix-generated code is not restricted to using IIOP and therefore contains generic code that is compatible with a multitude of transports.
- Artix maps WSDL types to Java using the mapping described in the JAX-RPC specification. The resulting types are very different from those generated by an IDL-to-Java compiler.

Generated files

The Artix code generator produces a number of files from the Artix contract. They are named according to the port name specified when the code was generated. The files include:

portTypeName.**java** defines the Java interface that both the client and server implement.

*portTypeName***Impl.java** defines the class used to implement the server.

*portTypeName***Server.java** is a simple main class for the server.

In addition to these files, the code generator also creates a class for each named schema type defined in the Artix contract. These files are named according to the type name they are given in the contract and contain the helper functions needed to use the data types. The naming convention for the helper type functions conforms to the JAX-RPC specification. For more information on using these generated data types see "Working with Artix Data Types" on page 45.

Generating code from the command line

You generate code at the command line using the command:

```
wsdltojava [-e service][-t port][-b binding][-i portType]
           [-d output_dir][-p [namespace=]package][-impl]
           [-server][-client][-types][-interface][-sample][-all]
           [-ant][-datahandlers][-nexclude namespace[=package]]
           [-ninclude namespace[=package]]artix-contract
```

You must specify the location of a valid Artix contract for the code generator to work. The default behavior of `wsdltojava` is to generate all of the java code needed to develop a client and server. You can also supply the following optional parameters to control the portions of the code generated:

`-e service`	Specifies the name of the service for which the tool will generate code. The default is to use the first service listed in the contract.
`-t port`	Specifies the name of the port for which code is generated. The default is to use the first port listed in the service.
`-b binding`	Specifies the name of the binding to use when generating code. The default is to use the first binding listed in the contract.
`-i portType`	Specifies the name of a portType for which code will be generated. You can specify this flag for each portType for which you want code generated. The default is to use the first portType in the contract.
`-d output_dir`	Specifies the directory to which the generated code is written. The default is the current working directory.
`-p [namespace=]package`	Specifies the name of the Java package to use for the generated code. You can optionally map a WSDL namespace to a particular package name if your contract has more than one namespace.
`-impl`	Generates the skeleton class for implementing the server defined by the contract.
`-server`	Generates a simple main class for the server.

`-client`	Generates only the Java interface and code needed to implement the complex types defined by the contract. This flag is equivalent to specifying `-interface -types`.
`-types`	Generates the code to implement the complex types defined by the contract.
`-interface`	Generates the Java interface for the service.
`-sample`	Generates a sample client that can be used to test your Java server.
`-all`	Generates code for all portTypes in the contract.
`-ant`	Generate an ant build target for the generated code.
`-datahandlers`	When a service uses SOAP w/ attachments as its payload format, generate code that uses `javax..activation.DataHandler` instead of the standard Java classes specified in the JAX-RPC specification. For more informaiton see "Using SOAP with Attachments" on page 111 and *Desinging Artix Solutions*.
`-nexclude` *`namespace[=package]`*	Instructs the code generator to skip the specified XMLSchema namespace when generating code. You can optionally specify a package name to use for the types that are not generated.
`-ninclude` *`namespace[=package]`*	Instructs the code generator to generate code for the specified XMLSchema namespace. You can optionally specify a package name to use for the types in the specified namespace.

Warning messages

If you generate code from a WSDL file that contains multiple `<portType>` elements, multiple bindings, multiple services, or multiple ports `wsdltojava` will generate a warning message informing you that it is using the first instance of each to use for generating code. If you use the command line flags to specify which instances to use, the warning message is not displayed.

Java Package Names

Artix packages

The Artix bus object which provides the transport and payload format independence in Artix is defined in the `com.iona.jbus` package. You will need to import this package and all of its subpackages into all Artix Java applications.

Generated type packages

The generated types are generated into a single package which must be imported for any methods using them. By default, the package name will be mapped from the target namespace of the schema describing the types. The default package name is created following the algorithm specified in the JAXB specification. The mapping algorithm follows four basic steps:

1. The leading `http://` or `urn://` are stripped off the namespace.
2. If the first string in the namespace is a valid internet domain, for example it ends in `.com` or `.gov`, the leading `www.` is stripped off the string, and the two remaining components are flipped.
3. If the final string in the namespace ends with a file extension of the pattern `.xxx` or `.xx`, the extension is stripped.
4. The remaining strings in the namespace are appended to the resulting string and separated by dots.
5. All letters are made lowercase.

For example, the XML namespace `http://www.widgetVendor.com/types/widgetTypes.xsd` would be mapped to the Java package name `com.widgetvendor.types.widgettypes`.

Java packages

Artix applications require a number of standard Java packages. These include:

javax.xml.namespace.QName provides the functionality to work with the XML QNames used to specify services.

javax.xml.rpc.* provides the APIs used to implement Artix Java clients. This package is not needed by server code.

java.io.* provides system input and output through data streams, serialization and the file system.

java.net.* provides the classes need to for communicating over a network. These classes are key to Artix applications that act as Web services.

Developing a Server

Overview

The Artix code generator generates server skeleton code and the implementation shell that serves as the starting point for developing an Artix-enabled server. The skeleton code hides the transport details, allowing you to focus on business logic.

Generating the server implementation class

The Artix code generation utility, `wsdltojava`, will generate an implementation class for your server when passed the `-impl` command flag.

Note: If your contract specifies any derived types or complex types you will also need to generate the code for supporting those types by specifying the `-types` flag.

Generated code

The implementation class code consists of two files:

PortName.**java** contains the interface the server implements.

*PortName***Impl.java** contains the class definition for the server's implementation class. It also contains empty shells for the methods that implement the operations defined in the contract.

Completing the server implementation

You must provide the logic for the operations specified in the contract that defines the server. To do this you edit the empty methods provided in *`PortName`*`Impl.java`. A generated implementation class for a contract defining a service with two operations, `sayHi` and `greetMe`, would resemble Example 7. Only the code portions highlighted in **`bold`** (in the bodies of the `greetMe()` and `sayHi()` methods) must be inserted by the programmer.

Example 7: *Implementation of the HelloWorld PortType in the Server*

```
// Java
import java.net.*;
import java.rmi.*;
```

Example 7: *Implementation of the HelloWorld PortType in the Server*

```
public class HelloWorldImpl {

    /**
     * greetMe
     *
     * @param: stringParam0 (String)
     * @return: String
     */
   public String greetMe(String stringParam0) {
        System.out.println("HelloWorld.greetMe() called with
   message: "+stringParam0);
        return "Hello Artix User: "+stringParam0;
    }

    /**
     * sayHi
     *
     * @return: String
     */
    public String sayHi() {
        System.out.println("HelloWorld.sayHi() called");
        return "Greetings from the Artix HelloWorld Server";
    }
```

Writing the server main()

The server `main()` of an Artix Java server must do three things before it can service requests:

1. Initialize the Artix bus.
2. Create a servant for the service implementation.
3. Register the server implementation with the Artix bus.
4. Start the Artix bus.

You can use `wsdltojava` to generate a server `main()` with the code to perform these steps by using the `-server` flag. The `main()` shown in Example 10 on page 18 was generated using `wsdltojava`.

Initializing the bus

The Artix bus is initialized using `com.iona.jbus.Bus.init()`. The method has the following signature:

```
static Bus init(String args[]);
```

`init()` takes the `args` parameter passed into the main as a required parameter. Optionally, you can also pass in a second string that specifies the name of the configuration scope from which the bus instance will read its runtime configuration.

This will create a bus instance to host your services, load the Artix configuration information for your application, and load the required plug-ins.

Before the bus can begin processing requests made on your server, you must register the servant object that implements your server's business logic with the bus. Registering the implementation object's servant with the bus allows the bus to create instances of the implementation object to service requests.

Creating a servant for your service implementation

Artix wraps service implementation objects in a `Servant` object that allows the bus to manage the object. To create a `com.iona.jbus.Servant` for your service implementation you create an instance of a `SingleInstanceServant` as shown in Example 8. The creator for a `SingleInstanceServant` uses the path of the WSDL file describing the service interface, an instance of your implementation object, and an instance of an initialized Artix bus. Example 8 shows the code to create a servant for the `HelloWorld` service.

Example 8: *Creating a Servant*

```
//Java
Servant servant =
  new SingleInstanceServant("./HelloWorld.wsdl",
                            new HelloWorldImpl(), bus);
```

Registering a servant for the server implementation

After creating the servant, you register it with the bus so that it can begin listening for requests. Servants are registered using the bus' `registerServant()` method. This registers the servant with a fixed address that is read from the contract associated with the application. The signature for `registerServant()` is shown in Example 9.

Example 9: *registerServant()*

```
void registerServant(Servant servant,
                     QName serviceName,
                     String portName)
throws BusException
```

In addition to the servant, `registerServant()` takes the service's QName as specified in the contract defining the service. You can also supply the name of the WSDL port you on which you want the servant activated. If no port name is given, the servant is activated on all ports.

Starting the bus

After the bus is initialized and the server implementation is registered with it, the bus is ready to listen for requests and pass them to the server for processing. To start the bus, you use the bus' `run()` method. Once the bus is started, it retains control of the process until it is shut down. The server's `main()` will be blocked until `run()` returns.

Completed server main()

Example 10 shows how the `main()` for a Java Artix server might look.

Example 10: *Server main()*

```
// Java
import com.iona.jbus.*;
import javax.xml.namespace.QName;

public class Server
{
  public static void main(String args[])
  throws Exception
  {
    // Initialize the Artix bus
    Bus bus = Bus.init(args);
```

Example 10: *Server main()*

```
    // Register the Servant
    QName name = new QName("http://xmlbus.com/HelloWorld",
                           "HelloWorldService");
    Servant servant =
                  new SingleInstanceServant("./HelloWorld.wsdl",
                                            new HelloWorldImpl());
    bus.registerServant(servant, name, "HelloWorldPort");

    // Start the Bus
    bus.run();
  }
}
```

Developing a Client

Overview

Artix Java clients are implemented using dynamic proxies as described in the JAX-RPC 1.1 specification. The interface used to create the proxy class is defined in the generated file *PortName*.**java**. The only Artix-specific code needed by an Artix Java client initializes and shuts down the Artix bus.

Initializing the bus

Client applications initialize the bus in the same manner as server applications, by calling the bus' `init()` method. Client applications, however, do not need to make a call to the bus' `run()` method.

Instantiating a service proxy

Artix Java clients use dynamic proxies, as described in the JAX-RPC specification, to make requests on servers. Dynamic proxies are created using the interface generated from your contract and the `javax.xml.rpc.Service` interface. You need the `QName` of the service for which you are creating the proxy, the `QName` of the endpoint with which the proxy will contact the service, and the URL of the contract defining the service. Once you have these three pieces of information, creating a dynamic proxy requires three steps:

1. Obtain an instance of `javax.xml.rpc.ServiceFactory` to create the service.

   Note: If your client is going to run inside of a J2EE container you will need to set the JAX-RPC `ServiceFactory` property to use the IONA `ServiceFactory` prior to getting the `ServiceFactory` object. You do this with the following code:

   ```
   System.setProperty("javax.xml.rpc.ServiceFactory",
      "com.iona.jbus.JBusServiceFactory");
   ```

2. Use the `ServiceFactory` to create a `Service` instance for the service to which the proxy will connect.
3. Use the `Service` to instantiate the dynamic proxy.

Obtaining a ServiceFactory instance

To obtain an instance of the `ServiceFactory` you call `ServiceFactory.newInstance()`. This returns the `ServiceFactory`. Only one is created per application and the same `ServiceFactory` is returned for each successive call.

Creating a Service instance

A `Service` instance is created from the `ServiceFactory` using `createService()`. `createService()` takes two arguments:

- the URL of the contract defining the service.
- the service's `QName`.

Creating the dynamic proxy

The dynamic proxy is created from the `Service` using `getPort()`. `getPort()` takes two arguments:

- the `QName` of the endpoint with which the proxy contacts the service.
- the name of the generated Java interface in *`PortName`*`.java` with `.class` appended. For example, if the generated interface's name is `HelloWorld`, this argument would be `HelloWorld.class`.

`getPort()` returns an instance of `java.rmi.Remote` that must be cast to the generated interface.

Shutting the bus down

Unlike a server that must shut down the bus from a separate thread, clients do not typically make a call to the bus' `run()` method and can simply call `shutdown()` on the bus before the main thread exits. It is advisable to pass `true` to `shutdown()` to ensure that the bus is fully shutdown before exiting.

Full client code

An Artix Java client developed to access `HelloWorldService` will look similar to Example 11.

Example 11: *Client Code*

```
//Java
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;

public class HelloWorldClient
{

  public static void main (String args[]) throws Exception
  {
1   Bus bus = Bus.init(args);

2   QName name = new QName("http://iona.com/HelloWorld",
                           "HelloWorldService");
3   QName portName = new QName("","HelloWorldPort");

4   String wsdlPath = "file:/./HelloWorld.wsdl";
    URL wsdlLocation = new File(wsdlPath).toURL();

5   ServiceFactory factory = ServiceFactory.newInstance();

6   Service service = factory.createService(wsdlLocation, name);

7   HelloWorld proxy = (HelloWorld)service.getPort(portName,
                                                   HelloWorld.class);

8   String string_out;

    string_out = proxy.sayHi();
    System.out.println(string_out);

9   bus.shutdown(true);

    }
}
```

The code does the following:

1. The `com.iona.jbus.Bus.init()` function initializes the bus.
2. Creates the service's `QName`.
3. Creates the `QName` of the endpoint with which the proxy will contact the service.
4. Creates the URL of the contract defining the service.
5. The `newInstance()` function returns the `ServiceFactory`.
6. The `createService()` function instantiates the `Service` from which the dynamic proxy is created.
7. The `getPort()` function returns a dynamic proxy to the `HelloWorld` service. `getPort()` returns an instance of `java.rmi.Remote` that must be cast to the interface defining the service.
8. Makes a call on the proxy to request service.
9. Shuts down the bus.

Building an Artix Application

Required jar files

Artix Java applications require that the following Artix jar files are in your classpath:

- *installdir*\lib\artix\java_runtime\2.1\it_bus-api.jar
- *installdir*\lib\artix\ws_common\2.1\it_wsdl.jar
- *installdir*\lib\artix\ws_common\2.1\it_ws_reflect.jar
- *installdir*\lib\artix\ws_common\2.1\it_ws_reflect_types.jar
- *installdir*\lib\common\ifc\1.1\ifc.jar
- *installdir*\lib\jaxrpc\jaxrpc\1.1\jaxrpc-api.jar

Other jar files

If your application uses SOAP with attachments, you will also need to include *install_dir*/lib/sun/activation/1.0.1/activation.jar on your classpath.

If your application uses xsd:any, you will need to include *install_dir*/lib/ws_common/2.1/saaj-api.jar on your classpath.

CHAPTER 3

Advanced Programming Issues

Several areas must be considered when programming complex Artix applications.

In this chapter

This chapter discusses the following topics:

Servant Registration	page 26
Proxy Creation	page 30
Getting a Bus	page 32
Threading	page 33
Setting Client Connection Attributes Using the Stub Interface	page 37
Class Loading	page 41

Servant Registration

Overview

In order to make a service accessible to remote client's, you must *register* its associated servant with a bus instance. Once the servant is registered with the bus instance the service is activated and begins listening for requests.

When a servant is instantiated in Java it is associated with the logical portion of an Artix contract. It is a Java instance of the interfaced defined in a WSDL `<portType>` element. At this point, a Java servant has no knowledge of the physical details of the service which it implements.

The servant is associated with the physical details of the service when it is registered with an instance of the Artix bus. At this point the servant is tied to the physical details defined by the WSDL `<port>` element defining the message format and transport used by the service.

Artix provides two methods for registering a servant:

Static registration ties the servant to a `<port>` element in the physical contract defining the service.

Transient registration ties the servant to a cloned `<port>` element.

In this section

This section discusses the following topics:

Static Servant Registration	page 27
Transient Servant Registration	page 28

Static Servant Registration

Overview

When a servant is registered as a *static* servant it is linked to a `<port>` definition that is read from the contract associated with the application. This means that a static servant is restricted to using a service from the fixed collection of services appearing in the contract.

Static servants are useful when a bus instance is only going to host a single instance of a servant. They are also useful when using references and you do not want to use the WSDL publishing plug-in because clients that have a copy of the service's contract have the servant's port information.

Registering

You register a static servant using the bus' `registerServant()` method. The signature for `registerServant()` is shown in Example 12.

Example 12: *registerServerFactory()*

```
void registerServant(Servant servant,
                     QName serviceName,
                     String portName)
throws BusException
```

In addition to the servant instance, `registerServant()` takes the service's QName as specified in the contract defining the service. You can also supply the name of the WSDL port you on which you want the servant activated. If no port name is given, the servant is activated on all ports. To register a servant on more than one specific port, you can call `registerServant()` multiple times and specifying a different port name on each call.

Example

Example 13 shows the code for registering a static servant.

Example 13: *Registering a Static Servant*

```
QName name = new QName("http://whoDunIt.com/Slueth",
                       "SluethService");
Servant servant = new SingleInstanceServant("./slueth.wsdl",
                                            new SluethImpl());
bus.registerServant(servant, name, "SluethHTTPPort");
```

Transient Servant Registration

Overview

When a servant is registered as a *transient* servant, Artix clones a `<service>` definition from the physical contract associated with the application and links the transient servant with the clone. This has the following effects:

- The transient servant's physical details are based on an existing `<service>` element that appears in the contract.
- The transient servant's service QName is replaced by a dynamically generated, unique service QName.
- The transient servant's addressing information is replaced such that each address is unique per-clone and per-port.

Transient servants are useful if the bus is going to be hosting a number of instances of a servant as when a service is a factory for other services.

Supported transports

While Artix will allow you to register any servant as transient, not all transports support the notion of transience. Currently, the only transports supported by Artix that can make use of transient servants are HTTP, CORBA, and IIOP.

Service templates

When using transient servants in your application, your contract must provide a *service template* for the servant. A service template is a WSDL service from which your transient servants will be cloned. When creating the service template for transient servants adhear to the following:

- The service template must come before any actual WSDL services defined in the contract. If you place your service templates after your actual WSDL service definitions, you may run into problems using the router.
- The service must use one of the supported transports.
- The service must fully describe the properties of the transport being used.
- The address specified for either a CORBA service or a IIOP service must be `ior:`. Specifying any other address in the template will cause the servants to have invalid IORs.

Registering

You register a transient servant using the bus' `registerTransientServant()` method. The signature of `registerTransientServant()` is shown in Example 14.

Example 14: *registerTransientServant()*

```
public abstract QName registerTransientServant(Servant servant,
                                               QName serviceName)
throws BusException;
```

In addition to the servant instance, `registerTransientServant()` takes the service's QName as specified in the contract defining the service. Unlike `registerServant()`, `registerTransientServant()` does not allow you to specify a port name because the bus dynamically assigns a port to the transient servant.

Transient servant QNames

Because the newly created transient servant is cloned from the service whose QName was supplied, the new servant has a different QName. The transient servant's QName is returned when you invoke `registerTransientServant()`. The returned QName is the QName you use when creating references for the transient servant or when destroying the transient servant.

Example

Example 15 shows the code for registering a transient servant.

Example 15: *Registering a Transient Servant*

```
QName name = new QName("http://whoDunIt.com/Slueth",
                       "SluethService");
Servant servant = new SingleInstanceServant("./slueth.wsdl",
                                            new SluethImpl());
QName transientName = bus.registerTransientServant(servant,
                                                   name,
                                                   "SluethHTTPPort");
```

Proxy Creation

Overview

While the Artix Java API's use dynamic proxies as specified by JAX-RPC, you may not always be able to use the JAX-RPC specified method for creating a service proxy. Artix provides a method for creating service proxies that bypasses the steps outlined in the JAX-RPC specification.

createClient()

You can create service proxies using the bus' `createClient()` method. `createClient()` takes the URL of the service's contract, the QName of the service, the name of the port the proxy will use to connect to the service, and the Java `Class` representing the service's remote interface and returns a JAX-RPC style dynamic proxy for the service if it is successful. `createClient()`'s signature is shown in Example 16.

Example 16: *Bus.createClient()*

```
Remote Bus.createClient(URL wsdlUrl, QName serviceName,
                        String portName, Class interfaceClass)
throws BusException
```

Example

Example 17 shows the code for creating a service proxy using `createClient()`.

Example 17: *Creating a Service Proxy using createClient()*

```
1  QName name = new QName("http://www.buystuff.com",
                          "RegisterService");

2  String portName = new String("RegisterPort");

3  String wsdlPath = "file:/./resister.wsdl";
   URL wsdlURL = new File(wsdlPath).toURL();

4  // Bus bus obtained earlier
   Register proxy = bus.createClient(wsdlURL, name, portName,
                                     Register.class);
```

The code in Example 17 does the following:

1. Creates the `QName` for the service from the contract defining the application. In this example, the service, `RegisterService`, is defined in the namespace `http:\\www.buystuff.com`.
2. Creates a `String` to hold the name of the `<port>` defining the transport the proxy will use to contact the service. In this example, the transport details are defined in a `<port>` named `RegisterPort`.
3. Creates a `URL` specifying where the service's contract can be located. In this example, the contract, `register.wsdl`, is located in the client's directory.
4. Calls `createClient()` with the correct parameters to create a service proxy for the `Register` service.

Getting a Bus

Overview

There are many instances where you need to get the defualt bus for an application. These include working with contexts and generating references. When you are in the mainline code of your application, you will have access to the instance of the bus you initialized. However, inside the implementation object of your service or in methods outside the scope of your client application's mainline.

Inside a service implementation object

If you are in a service's implementation object, you can use the code shown in Example 18.

Example 18: *Getting a Bus Reference Inside a Servant*

```
com.iona.jbus.Bus bus = DispatchLocals.getCurrentBus();
```

From a client proxy

If you have a client proxy object, you can use the JAX-RPC `Stub` interface as shown in Example 19.

Example 19: *Getting a Bus Reference from a Client Proxy*

```
Stub clientStub = (Stub)client;
com.iona.jbus.MessageContext context =
clientStub._getProperty(com.iona.jbus.MessageContext.ARTIX_
   MESSAGE_CONTEXT);
com.iona.jbus.Bus bus = context.getTheBus();
```

Threading

Overview

The Artix bus is a multithreaded C++ application that uses a thread pool to hand out threads. When using the Artix Java APIs, you can use the Artix configuration file to control how the C++ core manages its threads. In addition the Artix Java APIs provide three servant threading models to handle requests from the bus. These models are:

- single-instance multithreaded
- serialized single-instance
- per-invocation

Thread pool configuration

The bus's thread pool is configured in your applications configuration scope. This configuration scope is specified in the main Artix configuration file.

There are three configuration variables that are used to configure the bus' thread pool:

- `thread_pool:initial_threads` sets the number of initial threads in each port's thread pool.
- `thread_pool:low_water_mark` sets the minimum number of threads in each service's thread pool.
- `thread_pool:high_water_mark` sets the maximum number of threads allowed in each service's thread pool.

For a detailed discussion of Artix configuration see *Deploying and Managing Artix Solutions*.

Single-instance multithreaded servant

The standard Artix servant is the `SingleInstanceServant`. The `SingleInstanceServant` provides a multi-threaded, single instance usage model to the user. This means that all invocation threads for a given port access the same implementation object as shown in Figure 1 on page 34. The `SingleInstanceServant` provides no thread safety for the user code.

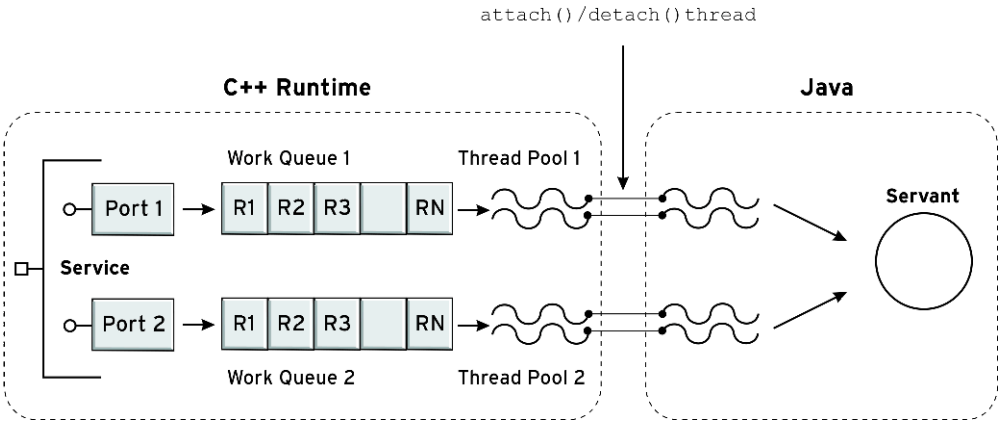

Figure 1: *SingleInstanceServant*

To instaniate a `SingleInstanceServant` you need to provide the path of the WSDL file describing the service interface, an instance of your implementation object, and an instance of an initialized Artix bus. Example 20 shows an example of instantiating a `SingleInstanceServant`.

Example 20: *Creating a SingleInstnaceServant*

```
//Java
Servant servant =
  new SingleInstanceServant(new HelloImpl(),
                            "./hello.wsdl", bus);
```

Serialized single-instance servant

Artix provides a thread safe single-instance servant called a `SerializedServant`. A `SerializedServant` ensures that all invocations are routed to a single implementation object in a serialized manner as shown in Figure 2 on page 35. Using a `SerializedServant` is equivalent to using a `SingleInstanceServant` whose target object is completely synchronized.

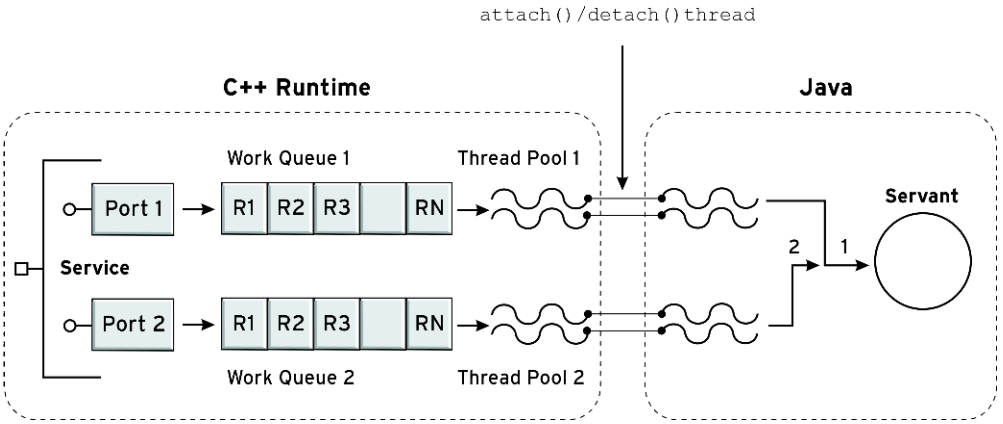

Figure 2: *SerializedServant*

To instantiate a `SerializedServant` you need to provide the path of the WSDL file describing the service interface, an instance of your implementation object, and an instance of an initialized Artix bus. Example 20 shows an example of instantiating a `SerializedServant`.

Example 21: *Creating a SerializedServant*

```
//Java
Servant servant = new SerializedServant(new HelloImpl(),
                                        "./hello.wsdl", bus);
```

Per-invocation servant

In addition to the multithreaded single instance servants, Artix provides a per-invocation servant. This servant is implemented by the `PerInvocationServant` class. A `PerInvocationServant` guarantees that a

separate instance of the implementation object will be used for each invocation as shown in Figure 3 on page 36. This ensures thread safety, but does not allow the implementation object to have any statefull information.

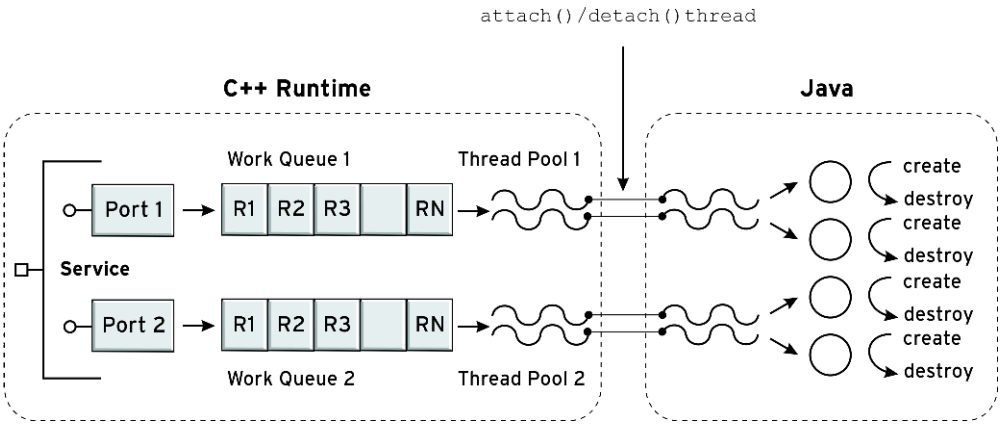

Figure 3: *PerInvocationServant*

To use a `PerInvocationServant`, your implementation object must either have a no-argument constructor, or implement the `Cloneable` interface and provide a `clone()` method. Like the other servants the `PerInvocationServant` needs the path of the WSDL file describing the service interface, an instance of your implementation object, and an instance of an initialized Artix bus when being instantiated. Example 22 shows the code for instantiating a `PerInvocationServant`.

Example 22: *Creating a PerInvocationServant*

```
//Java
Servant servant = new PerInvocationServant(new HelloImpl(),
                                           "./hello.wsdl", bus);
```

Setting Client Connection Attributes Using the Stub Interface

Overview

The JAX-RPC specification lists four standard properties to which a service proxy's `Stub` interface provides access. Artix provides support for setting three of them:

- Username
- Password
- Endpoint Address

Currently, Artix only supports setting these properties for HTTP connections.

The Stub interface

As required by the JAX-RPC specification, all Artix proxies implement the `javax.xml.rpc.Stub` interface. This interface provides access to a number of low-level properties used in connecting the proxy to the service implementation. To access these low-level properties the Stub interface has two methods:

- `_getProperty()` returns the value of the specified property.
- `_setProperty()` allows you to set the value of the specified property.

Getting a Stub object

Because all Artix proxies implement the `Stub` interface, you can simply cast an Artix proxy to a `Stub` object. Example 23 shows code getting a `Stub` object from an Artix proxy.

Example 23: *Casting a Client Proxy to a Stub*

```
//Java
import javax.xml.rpc.*;

// client proxy, client, created earlier
Stub clientStub = (Stub) client;
```

Setting the username property

One of the standard properties specified in the JAX-RPC specification is the `javax.xml.rpc.security.auth.username` property. It is used to set a username for use in basic authentication systems. Artix uses this property to set the HTTP transport's `UserName` property.

To set the username property using the client's `Stub` interface do the following:

1. Get a `Stub` object by casting your service proxy to a `Stub` as shown in Example 23 on page 37.
2. Create a `String` containing the username for the value of the property.
3. Call `_setProperty()` on the `Stub` specifying `Stub.USERNAME_PROPERTY` as the property name and the `String` created in step 2 as the value of the property.

Example 24 on page 38 shows code for setting the username for a client.

Example 24: *Setting the Username Property on a Stub*

```
//Java
import javax.xml.rpc*

// Service proxy, secClient, obtained earlier
Stub secStub = (Stub)secClient;
String userName = new String("Smart");
secStub._setProperty(Stub.USERNAME_PROPERTY, userName);
```

Setting the password property

One of the standard properties specified in the JAX-RPC specification is the `javax.xml.rpc.security.auth.password` property. It is used to set a password for use in basic authentication systems. Artix uses this property to set the HTTP transport's `Password` property.

To set the username property using the client's `Stub` interface do the following:

1. Get a `Stub` object by casting your service proxy to a `Stub` as shown in Example 23 on page 37.
2. Create a `String` containing the password for the value of the property.
3. Call `_setProperty()` on the `Stub` specifying `Stub.PASSWORD_PROPERTY` as the property name and the `String` created in step 2 as the value of the property.

Example 25 on page 39 shows code for setting the password for a client.

Example 25: *Setting the Password Property on a Stub*

```
//Java
import javax.xml.rpc*

// Service proxy, secClient, obtained earlier
Stub secStub = (Stub)secClient;
String password = new String("86");
secStub._setProperty(Stub.PASSWORD_PROPERTY, password);
```

Setting the endpoint address

One of the standard properties specified in the JAX-RPC specification is the `javax.xml.rpc.service.endpoint.address` property. It is used to set the address for the target service. The property takes a `String` containing a valid HTTP URL that points to a service implementing the interface supported by the proxy.

You can only set this property before you invoke any of the service proxy's methods. Once the proxy makes a request on the remote service an HTTP service connection is established between the client and the service. Due to the multi-threaded nature of the Artix bus and the nature of HTTP connections, this connection cannot be broken and reassigned to a new endpoint. Attempts to reset the endpoint address property after invoking one of the proxy's methods will be ignored.

To set the endpoint address property using the client's `Stub` interface do the following:

1. Get a `Stub` object by casting your service proxy to a `Stub` as shown in Example 23 on page 37.
2. Create a `String` containing the target endpoint's HTTP URL for the value of the property.
3. Call `_setProperty()` on the `Stub` specifying `Stub.ENDPOIT_PROPERTY` as the property name and the `String` created in step 2 as the value of the property.

Example 25 on page 39 shows code for setting the endpoint address property for a client.

Example 26: *Setting the Endpoint Address Property on a Stub*

```
//Java
import javax.xml.rpc*

// Service proxy, secClient, obtained earlier
Stub secStub = (Stub)secClient;
String endpt = new
               String("http://control.silencecone.net/9986");
secStub._setProperty(Stub.ENDPOINT_PROPERTY, endpt);
```

Class Loading

Overview

There may be occasions where the jars provided with Artix conflict with the jars used in your environment. In particular, you may be using different versions of the Xerces XML parser and Log4J. To handle such situations, Artix provides a class loader firewall that isolates the Artix runtime class loader from the application class loader and the system class loader. This allows the Artix runtime to load the jars it needs and your application to load your versions of any jars that conflict.

How the class loader firewall works

The class loader firewall provides a mechanism for your to hide the application class loader's jar files from the Artix runtime. It does this by exposing a simple mechanism for you to create a set of positive filters defining what classes loaded by the application class loader are visible to the Artix runtime's class loader and specifying the location from which the Artix runtime class loader will load its classes. Any classes not matched by a positive filter are blocked from the Artix runtime's class loader and will only be loaded from the locations specified in the firewall's configuration file. Figure 4 shows how the class loader firewall blocks off the Artix runtime.

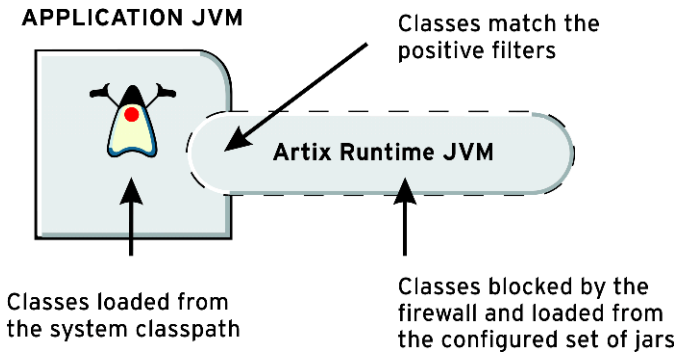

Figure 4: *Class Loader Firewall*

For example, in most cases you would create a positive filter allowing all of the J2SE classes into the Artix runtime. However, you would not create a positive filter for the Xerces classes if your applications use a different version of Xerces than Artix does. Artix will need to load its own Xerces classes in order to operate.

Configuring the firewall class loader

To use the class loader firewall with an Artix Java application do the following:

1. Create a file called `artix_ce.xml` and place it in your application's classpath.
2. Using the `artix_ce.xml` file included with the Java firewall demo as a template, define the filters to only allow the desired packages from the Artix class loader to be visible to your application code.
3. Define the rules governing where the Artix class loader will look for specific classes in the `<ce:loader>` element of `artix_ce.xml`.

Defining class filters

The class loader firewall, if it finds an `artix_ce.xml` file in the classpath, assumes that all classes not specified by a positive filter are to be blocked from the Artix runtime's class loader. You define positive filters using one of two `<ce:filter>` attributes: `type="discover"` and `type="pattern"`.

Using type="discover"

The discover filter type specifies that the class loader will discover the filters from the location specified in the `discover-source` attribute. Table 1 shows the values for `discover-source`.

Table 1: *discover-source values for the Class Loader Firewall*

Value	Meaning
`jre`	Discover the filters need to load all of the classes for the currently running JRE. It is highly recommended that this filter is included in your `artix_ce.xml` definition.

Table 1: *discover-source values for the Class Loader Firewall*

Value	Meaning
`jar`	Discover the filters to load all of the classes from the specified jar file. Jar file locations can be given using relative or absolute file names. For example to load all of the classes in `myApp.jar`, you could define a filter like `<ce:filter type="discover" discover-source="jar">.\myApp.jar</ce:filter>`.
`jar-of`	Discover the filters needed to load specified resource. This option makes it possible to discover the contents of jar files which you know are reachable through the class loading system, but which you do not know the actual location. Resources can be classes, properties files, or HTML files. For example to load the libraries for the `EJBHome` class, you could use a filter like `<ce:filter type="discover" discover-source="jar-of">javax/ejb/EJBHome.class</ce:filter>`.

Using type="pattern"

The pattern filter type directly specifies a package pattern to be allowed through the firewall from the application's class loader. The syntax for specifying package patterns is similar to the syntax used in Java `import` statements. For example, to specify that all classes from `javax.xml.rpc` are to be allowed through the firewall you could use a filter like `<ce:filter type="pattern">javax.xml.rpc.*</ce:filter>`. You could also drop the asterisk(*) and use the filter `<ce:filter type="pattern">javax.xml.rpc.</ce:filter>`.

Defining negative filters

Occasionally a positive filter will allow classes that you want blocked from the Artix runtime class loader to be visible through the firewall. This is particularly true with `com.iona.jbus`. The Artix runtime needs to share a number of resources from this package with the application code, but it also needs to ensure that some of its resources are loaded from the Artix jar files.

To solve this problem the class loader firewall allows you to define negative filters. To define a negative filter you use a value of `negative-pattern` for the `type` attribute of the filter. This tells the firewall to block any resources that match the pattern specified. For example, to block the system's

JAX-RPC classes from being loaded into the Artix runtime you could define a filter like `<ce:filter type="negative-pattern">com.iona.jbus.jaxrpc.<\ce:filter>`.

Specifying the location for loading blocked resources

The location from which the Artix runtime class loader will load resources blocked by the firewall are specified in the `<ce:loader>` element of `artix_ce.xml`. Inside the loader definition, you use a number of `<ce:location>` elements to specify the location of specific resources. These locations can be either the relative or absolute pathnames of a jar file. You can also specify a directory in which the class loader will search for the required jar files.

For example, if all of your Artix specific jar files are stored in the location in which they were installed you could use a loader element similar to Example 27 to specify the proper Xerces and Log4J version to load into the Artix runtime.

Example 27: *Loader Definition to Load Xerces and Log4J*

```
<ce:loader>
  <ce:loaction>C:\IONA\lib\apache\jakarta-log4j\1.2.6\log4j.jar<\ce:loaction>
  <ce:location>C:\IONA\lib\apache\xerces\2.5.0\xercesImpl.jar<\ce:location>
</ce:loader>
```

Examples

For an example of using the Artix class loader firewall see the `java_firewall` demo in the `demos\basic` folder of your Artix installation. The demo provides an example of using the class loader firewall to shield the Artix runtime from different versions of Xerces and Log4J.

CHAPTER 4

Working with Artix Data Types

Artix maps XMLSchema data types in an Artix contract into Java data types. For XMLSchema simple types the mapping is a one-to-one mapping to Java primitive types. For complex types, Artix follows the JAX-RPC specification for mapping complex types into Java objects.

In this chapter

This chapter discusses the following topics:

Using Native XMLSchema Simple Types	page 47
Defining Your Own Simple Types	page 53
Using XMLSchema Complex Types	page 56
Using XMLSchema any Elements	page 84
SOAP Arrays	page 92
Lists	page 95
Enumerations	page 98
Deriving Types Using <complexContent>	page 104
Holder Classes	page 107

Using SOAP with Attachments page 111

Using Native XMLSchema Simple Types

Overview

Artix follows the JAX-RPC specification for mapping native XMLSchema types into Java. In most cases, the mapping from a native XMLSchema type is to a primitive Java type. However, some instances require a more complex mapping.

In this section

This section contains the following subsections:

Simple Type Mapping	page 48
Special Simple Type Mappings	page 50
Unsupported Simple Types	page 52

Simple Type Mapping

Overview

When a message part is described as being of one of the simple XMLSchema types, the generated parameter's type will be of a corresponding primitive Java type. For example, the message description shown in Example 28 will cause a parameter, `score`, of type `int` to be generated.

Example 28: *Message Description Using a Simple Type*

```
<message name="scoreResponse">
  <part name="score" type="xsd:int" />
</message>
```

Table of simple type mappings

The simple type mappings are shown in Table 2.

Table 2: *Simple Schema Type to Primitive Java Type Mapping*

Schema Type	Java Type
`xsd:string`	`java.lang.String`
`xsd:int`	`int`
`xsd:insignedInt`	`long`
`xsd:long`	`long`
`xsd:unsignedLong`	`java.math.BigInteger`
`xsd:short`	`short`
`xsd:unsignedShort`	`int`
`xsd:float`	`float`
`xsd:double`	`double`
`xsd:boolean`	`boolean`
`xsd:byte`	`byte`
`xsd:integer`	`java.math.BigInteger`

Table 2: *Simple Schema Type to Primitive Java Type Mapping*

Schema Type	Java Type
xsd:positiveInteger	java.math.BigInteger
xsd:negativeInteger	java.math.BigInteger
xsd:nonPositiveInteger	java.math.BigInteger
xsd:nonNegativeInteger	java.math.BigInteger
xsd:decimal	java.math.BigDecimal
xsd:dateTime	java.util.Calendar
xsd:QName	javax.xml.namespace.QName
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:ID	java.lang.String
xsd:anySimpleType	java.lang.String
xsd:anyURI	java.lang.String
xsd:gYear	java.lang.String
xsd:gMonth	java.lang.String
xsd:gDay	java.lang.String
xsd:gYearMonth	java.lang.String
xsd:gMonthDay	java.lang.String

Simple type validation

Artix Java validates XMLSchema simple types when they are passed to the bus for writing to the wire. This means that when you are working with data elements that are mapped from XMLSchema simple types you should take care to ensure that they conform to the restrictions of the XMLSchema type. For example, the Java APIs would allow you to set a value of `-10` into a data element that is mapped to an `xsd:positiveInteger`. However, when the bus attempted to write out the message containing that data element, the bus would throw an exception.

Special Simple Type Mappings

Overview

Mapping XMLSchema simple types to Java primitives does not work for all possible data descriptions in an Artix contract. Several cases require that an XMLSchema simple type is mapped to the Java primitive's corresponding wrapper type. These cases include:

- an `<element>` with its `nillable` attribute set to `true` as shown in Example 29.

  Example 29: *Nillable Element*

  ```
  <element name="finned" type="xsd:boolean" nillable="true" />
  ```

- an `<element>` with its `minOccurs` attribute set to `0` and its `maxOccurs` attribute set to `1` or its `maxOccurs` attribute not specified as shown in Example 30.

  Example 30: *minOccurs set to Zero*

  ```
  <element name="plane" type="xsd:string" minOccurs="0" />
  ```

- an `<attribute>` with its `use` attribute set to `optional`, or not specified, and having neither its `default` attribute nor its `fixed` attribute specified as shown in Example 31.

  Example 31: *Optional Attribute Description*

  ```
  <element name="date">
    <complexType>
      <sequence/>
      <attribute name="calType" type="xsd:string"
       use="optional" />
    </complexType>
  </element>
  ```

Mappings

Table 3 shows how XMLSchema simple types are mapped into Java wrapper classes in these special cases.

Table 3: *simple Schema Type to Java Wrapper Class Mapping*

Schema Type	Java Type
`xsd:int`	`java.lang.Integer`
`xsd:long`	`java.lang.Long`
`xsd:short`	`java.lang.Short`
`xsd:float`	`java.lang.Float`
`xsd:double`	`java.lang.Double`
`xsd:boolean`	`java.lang.Boolean`
`xsd:byte`	`java.lang.Byte`

Unsupported Simple Types

List of unsupported simple types

The following XMLSchema simple types are currently not supported by Artix Java:

```
xsd:duration
xsd:time
xsd:date
xsd:ENTITY
xsd:NOTATION
xsd:IDREF
soapenc:base64
```

Defining Your Own Simple Types

Overview

XMLSchema allows you to create simple types by deriving a new type from another primitive type or simple type. Simple types are described in the <types> section of an Artix contract using a <simpleType> element.

The new types are described by restricting the *base type* with one or more of a number of facets. These facets limit the possible valid values that can be stored in the new type. For example, you could define a simple type, SSN, which is a string of exactly 9 characters. Each of the primitive XMLSchema types has their own set of optional facets. Artix does not enforce the use of all the possible facets. However, to ensure interoperability, your service should enforce any restrictions described in the contract.

Procedure

To define your own simple type do the following:

1. Determine the base type for your new simple type.
2. Based on the available facets for the chosen base type, determine what restrictions define the new type.
3. Using the syntax shown in this section, enter the appropriate <simpleType> element into the <types> section of your contract.

Describing a simple type in XMLSchema

Example 32 shows the syntax for describing a simple type.

Example 32: *Simple Type Syntax*

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value"/>
    <facet value="value"/>
    ...
  </restriction>
</simpleType>
```

The type description is enclosed in a <simpleType> element and identified by the value of the name attribute. The base type from which the new simple type is being defined is specified by the base attribute of the <restriction>

element. Each facet element is specified within the `<restriction>` element. The available facets and their valid setting depends on the base type. For example, `xsd:string` has six facets including:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `whitespace`

Example 33 shows an example of a simple type, `SSN`, which represents a social security number. The resulting type will be a string of the form `XXX-XX-XXXX`. `<SSN>032-43-9876<SSN>` is a valid value, but `<SSN>032439876</SSN>` is not valid.

Example 33: *SSN Simple Type Description*

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

Mapping simple types to Java

Artix maps user-defined simple types to the Java type of the simple type's base type. So any message using the simple type `SSN`, shown in Example 33, would be mapped to a `String` because the base type of `SSN` is `xsd:string`. For example, the contract fragment shown in Example 34 would result in a Java method, `creditInfo()`, which took a parameter, `socNum`, of `String`.

Example 34: *Credit Request with Simple Types*

```
<message name="creditRequest">
  <part name="socNum" type="SSN" />
</message>
...
<portType name="creditAgent">
  <operation name="creditInfo">
    <input message="tns:creditRequest" name="credRec" />
    <output message="tns:creditReport" name="credRep" />
  </operation>
</portType>
```

Because this mapping does not place any restrictions on the values placed a variable that is mapped from a simple type and Artix does not enforce all facets, you must ensure that your application logic enforces the restrictions described in the contract for maximum interoperability.

Unenforced facets

Artix does not enforce the following facets:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `whiteSpace`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`
- `totalDigits`
- `fractionDigits`

Enforced facets

Artix enforces the following facets:

- `enumeration`

For more information on the enumeration facet, read "Enumerations" on page 98.

Using XMLSchema Complex Types

Overview

Complex types are described in the `<types>` section of an Artix contract. Typically, they are described in XMLSchema using a `<complexType>` element. In contrast to simple types, complex types can contain multiple elements and have attributes.

Complex types are generated into Java objects according to the mapping specified in the JAX-RPC specification. Each generated object has a default constructor, methods for setting and getting values from the object, and a method for stiringifying the object.

In this section

This section contains the following subsections:

Sequence and All Complex Types	page 57
Choice Complex Types	page 64
Attributes	page 68
Nesting Complex Types	page 72
Deriving a Complex Type from a Simple Type	page 78
Occurrence Constraints	page 81

Sequence and All Complex Types

Overview

Complex types often describe basic structures that contain a number of fields or elements. XMLSchema provides two mechanisms for describing a structure. One method is to describe the structure inside of a <sequence> element. The other is to describe the structure inside of an <all> element. Both methods of describing a structure result in the same generated Java classes.

The difference between using a <sequence> and an <all> is in how the elements of the structure are passed on the wire. When a structure is described using a <sequence>, the elements are passed on the wire in the exact order they are specified in the contract. When the structure is described using an <all>, the elements of the structure can be passed on the wire in any order.

Note: If neither <sequence>, <all>, nor <choice> is used to specify how the elements of the complex type are to be transmitted, the default is <sequence>.

Mapping to Java

A complex type described with <sequence> or with <all> is mapped to a Java class whose name is derived from the name attribute of the <complexType> element in the contract from which the type is generated. As specified in the JAX-RPC specification, the generated class has a getter and setter method for each element described in the type. The individual elements of the complex type are mapped to private variables within the generated class.

The generated setter methods are named by prepending `set` onto the name of the element as given in the contract. They take a single parameter of the type of the element and have no return value. For example, if a complex type contained the element shown in Example 35, the generated setter method would have the signature `void setName(String val)`.

Example 35: *Element Name Description*

```
<complexType name="Address">
  <all>
    <element name="Name" type="xsd:string" />
    ...
  </all>
</complexType>
```

The generated getter methods are named by prepending get onto the name of the element as given in the contract. They take no parameters and return the value of the specified element. For example, the generated getter method for the element described in Example 35 would have the signature `String getName()`.

Note: If the name of the element begins with a lowercase letter, the getter and setter methods will capitalize the first letter of the element name before prepending `get` or `set`.

In addition to the getter and setter methods, Artix also generates a `toString()` method for each complex type. The `toString()` method returns a string containing a labeled list of the values for each element in the class.

The maxOccurs attribute

Any elements whose `maxOccurs` attribute is set to a value greater than one or set to `unbounded`, results in the generation of a Java array to contain the value of the element. For example, the element described in Example 36 would result in the generation of a private variable, `observedSpeed`,of type `float[]`.

Example 36: *Element with MaxOccurs Greater than One*

```
<complexType name="drugTestResults">
  <sequence>
    <element name="observedSpeed" type="xsd:float"
             maxOccurs="unbounded"/>
...
  </sequence>
</complexType>
```

The getter and setter methods for `observedSpeed` are shown in Example 37.

Example 37: *observedSpeed Getter and Setter Methods*

```
// Java
public class drugTestResults
{
  private float[] observedSpeed;
...
  void setObservedSpeed(float[] val);
  float[] getObservedSpeed();
...
}
```

Example

Suppose you had a contract with the complex type, `monsterStats`, shown in Example 38.

Example 38: *monsterStats Description*

```
<complexType name="monsterStats">
  <all>
    <element name="name" type="xsd:string" />
    <element name="weight" type="xsd:long" />
    <element name="origin" type="xsd:string" />
    <element name="strength" type="xsd:float" />
    <element name="specialAttack" type="xsd:string"
             maxOccurs="3" />
  </all>
</complexType>
```

The Java class generated to support `monsterStats` would be similar to Example 39.

Example 39: *monsterStats Java Class*

```
// Java
public class monsterStats
{
  public static final String TARGET_NAMESPACE =
   "http://monsterBootCamp.com/types/monsterTypes";

  private String name;
  private long weight;
  private String origin;
  private float strength;
  private String[] specialAttack;

  public void setName(String val)
  {
    name=val;
  }
  public String getName()
  {
    return name;
  }
```

Example 39: *monsterStats Java Class*

```
public void setWeight(long val)
{
  weight=val;
}
public long getWeight()
{
  return weight;
}

public void setOrigin(String val)
{
  origin=val;
}
String getOrigin()
{
  return origin;
}

public void setStrength(float val)
{
  strength=val;
}
public float getStrength()
{
  return strength;
}

public void setSpecialAttack(String[] val)
{
  specialAttack=val;
}
public String[] getSpecialAttack()
{
  return specialAttack;
}
```

Example 39: *monsterStats Java Class*

```
public void setWeight(long val)
{
  weight=val;
}
public long getWeight()
{
  return weight;
}

public void setOrigin(String val)
{
  origin=val;
}
String getOrigin()
{
  return origin;
}

public void setStrength(float val)
{
  strength=val;
}
public float getStrength()
{
  return strength;
}

public void setSpecialAttack(String[] val)
{
  specialAttack=val;
}
public String[] getSpecialAttack()
{
  return specialAttack;
}
```

Example 39: *monsterStats Java Class*

```
  public String toString()
  {
        StringBuffer buffer = new StringBuffer();
        if (name != null) {
            buffer.append("name: "+name+"\n");
        }
        if (weight != null) {
            buffer.append("weight: "+weight+"\n");
        }
        if (origin != null) {
            buffer.append("origin: "+origin+"\n");
        }
        if (strength != null) {
            buffer.append("strength: "+strength+"\n");
        }
        if (specialAttack != null) {
            buffer.append("specialAttack: "+specialAttack+"\n");
        }
        return buffer.toString();
    }
}
```

Choice Complex Types

Overview

XMLSchema allows you to describe a complex type that may contain any one of a number of elements. This is done using a <choice> element as part of the complex type description. When elements are contained within a <choice> element, only one of the elements will be transmitted across the wire.

Mapping to Java

Like complex types described with a <sequence> element or with an <all> element, complex types described with a <choice> element are mapped to a Java class with getter and setter methods for each possible element inside the <choice> element. In addition, the generated Java class for a <choice> complex type includes an additional element, `_discriminator`, to hold the *discriminator* and a method for each element to determine if it is the current valid value for the choice. For each element in the choice, a method `isSet`*`elem_name`*`()` is generated. If the element is the currently valid value, its `isSet` method returns `true`. If not, the method returns `false`.

The discriminator is set in each of the complex type elements' setter methods. This means that while any of the elements in the Java object representing the complex type may contain valid data, the discriminator points to the last element whose value was set. As stated in the Web services specification only the element to which the discriminator is set will be placed on the wire by a server. For Artix developers this has two implications:

1. Artix servers will only write out the value for the last element set on an object representing a <choice> complex type.
2. When Artix clients receive an object representing a <choice> complex type, only the element pointed to by the discriminator will contain valid data.

Example

Suppose you had a contract with the complex type, `terrainReport`, shown in Example 40.

Example 40: *terrainReport Description*

```
<complexType name="terrainReport">
  <choice>
    <element name="water" type="xsd:float" />
    <element name="pier" type="xsd:short" />
    <element name="street" type="xsd:long" />
  </choice>
</complexType>
```

The Java class generated to represent `terrainReport` would be similar to Example 41.

Example 41: *terrainReport Java Class*

```
// Java
public class TerrainReport
{
  public static final String TARGET_NAMESPACE =
   "http://GlobeStrollers.com";

  private String __discriminator;

  private float water;
  private short pier;
  private long street;
```

Example 41: *terrainReport Java Class*

```
public void setWater(float _v)
{
  this.water=_v;
  _discriminator="water"'
}
public float getWater()
{
  return water;
}
public boolean isSetWater()
{
  if(__discriminator != null &&
     __discriminator.equals("water")) {
  return true;
  }

  return false;
}

public void setPier(short _v)
{
  this.pier=_v;
  _discriminator="pier";
}
public short getPier()
{
  return pier;
}
public boolean isSetPier()
{
  if(__discriminator != null &&
     __discriminator.equals("pier")) {
  return true;
  }

  return false;
}
```

Example 41: *terrainReport Java Class*

```
  public void setStreet(long _v)
  {
    this.street=_v;
    _discriminator="street";
  }
  public long getStreet()
  {
    return street;
  }
  public boolean isSetStreet()
  {
    if(__discriminator != null &&
       __discriminator.equals("street")) {
    return true;
    }

    return false;
  }

  public void _setToNoMember()
  {
    __discriminator = null;
  }

  public String toString()
  {
        StringBuffer buffer = new StringBuffer();
        if (water != null) {
            buffer.append("water: "+water+"\n");
        }
        if (pier != null) {
            buffer.append("pier: "+pier+"\n");
        }
        if (street != null) {
            buffer.append("street: "+street+"\n");
        }
        return buffer.toString();
    }
}
```

Attributes

Overview

Artix supports the use of <attribute> declarations within the scope of a <complexType> definition. When defining structures for an XML document <attribute> declarations provide a means of adding information to be specified within the tag, not the value that the tag contains. In other words, when describing the XML element <value currency="euro">410<\value> in XMLSchema currency would be described using an <attribute> declaration as shown in Example 42.

Example 42: *XMLSchema for value*

```
<element name="value">
  <complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="currency" type="xsd:string"
                       use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

When describing data types for use in developing application logic, however, attributes are treated as elements of a structure. For each <attribute> declaration contained within a complex type description, an element is generated in the class for the attribute along with the appropriate getter and setter methods. The application code must respect the use attribute of the attribute, but the generated Java code does not enforce this behavior.

Describing an attribute in XMLSchema

An XMLSchema <attribute> declaration has two required attributes. The name attribute identifies the attribute. The use attribute specifies if the attribute is required, optional, or prohibited..

An <attribute> declaration also has two optional attributes. The type attribute specifies the type of value the attribute can take. It is used when the attribute takes a value of a primitive type or of a type that is predefined in the contract. If the type attribute is omitted from the <attribute> declaration, the format of the data value must be described as part of the

<attribute> declaration. Example 43 shows an <attribute> declaration for an attribute, catagory, that can take the values autobiography, non-fiction, or fiction.

Example 43: *Attribute with an In-Line Data Description*

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

Example 44 shows an alternate description of the catagory attribute using the type attribute.

Example 44: *Category Attribute Using the type Attribute*

```
<simpleType name="catagoryType">
  <restriction base="xsd:string">
    <enumeration value="autobiography"/>
    <enumeration value="non-fiction"/>
    <enumeration value="fiction"/>
  </restriction>
</simpleType>
<complexType name="attributed">
...
  <attribute name="category" type="catagoryType" use="required">
</complexType>
```

The default/fixed attribute can be used when the use attribute is set to optional. When the default attribute is given, the value of the generated element is defaulted to the value specified. When the fixed attribute is given, the value of the generated element is set to the value specified and cannot be changed. In the generated Java class, using the fixed attribute results in the generated element not having a setter method.

Example mapping to Java

Suppose you had a contract with the complex type, `terrainReport`, shown in Example 45.

Example 45: *techDoc Description*

```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  <all>
  <attribute name="usefullness" type="xsd:float" use="optional"
             default="0.01" />
</complexType>
```

The Java class generated to represent `terrainReport` would be similar to Example 46.

Example 46: *techDoc Java Class*

```
// Java
public class TechDoc
{
  public static final String TARGET_NAMESPACE =
   "http://www.docUSA.org/usability";

  private String product;
  private short version;
  private Float usefullness = new Float(0.01);

  public void setProduct(String val)
  {
    product=val;
  }
  public String getProdcut()
  {
    return product;
  }
```

Example 46: *techDoc Java Class*

```
  public void setVersion(short val)
  {
    version=val;
  }
  public short getVersion()
  {
    return version;
  }

  public void setUsefullness(Float val)
  {
    usefullness=val;
  }
  public Float getUsefullness()
  {
    return usefullness;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();

    if (prudcut != null) {
      buffer.append("product: "+product+"\n");
    }
    if (version != null) {
      buffer.append("version: "+version+"\n");
    }
    if (usefullness != null) {
      buffer.append("usefullness: "+usefullness+"\n");
    }
  return buffer.toString();
  }
}
```

Nesting Complex Types

Overview

XMLSchema allows you to define complex types that contain elements of a complex type through a process called nesting. There are two ways of nesting complex types:

- Nesting with Named Types
- Nesting with Anonymous Types

Nesting with Named Types

When you nest with a named type your element declaration is the same as when the element was of a primitive type. The name of the complex type that describes the element's data is placed in the element's `type` attribute as shown in Example 47.

Example 47: *Nesting with a Named Type*

```
<complexType name="tweetyBird">
  <sequence>
    <element name="caged" type="xsd:boolean" />
    <element name="granny_proximity" type="xsd:int" />
  </sequence>
</complexType>
<complexType name="sylvesterState">
  <sequence>
    <element name="hunger" type="xsd:int" />
    <element name="food" type="tweetyBird" />
  </sequence>
</complexType>
```

The complex type `sylvesterState` includes an element, `food`, of type `tweetyBird`. The advantage of using named types is that `tweetyBird` can be reused as either a standalone complex type or nested in another complex type description.

Nesting with Anonymous Types

When you nest with an anonymous type, the element declaration for the nested complex type does not have a `type` attribute. Instead, the element's type description is provided as part of the element's declaration. Example 48 shows a description of `sylvesterState` using an anonymous type.

Example 48: *Nesting with an Anonymous Type*

```
<complexType name="sylvesterState">
  <sequence>
    <element name="hunger" type="xsd:int" />
    <element name="food">
      <complexType>
        <sequence>
          <element name="caged" type="xsd:boolean" />
          <element name="granny_proximity" type="xsd:int" />
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

In this example, the `food` element of `sylvesterState` still contains a `caged` sub-element and a `granny_proximity` sub-element. However, the complex type used to describe `food` cannot be re-used.

Mapping to Java

When a complex type containing nested complex types is mapped to Java, each complex type that is nested creates a generated class to represent it. The generated class for the top level complex type will have elements whose elements are instances of the class generated to represent their type. For example, the `sylvesterState` complex type, causes two Java classes to be generated. One to represent the type of the `food` element and one to represent `sylvesterState`.

The name of the classes generated to support the nested complex types depends on the style of nesting used. For named nested complex types, the generated class takes its name from the `name` attribute of the complex type used to describe it. So the nested type in Example 47 on page 72 would result in the generation of a class called `TweetyBird`. The `food` element of `SylvesterState` would be an instance of `TweetyBird`.

When you use anonymous nested complex types Artix names the class generated to represent the nested class by appending `_type` to the name of the parent complex type's `name` attribute. If that does not produce a unique name, Artix will append *_n*, where *n* is an incrementing whole number, to the name until it finds a unique name for the generated class. For example, the nested type in Example 48 on page 73 would generate a class, `SylvesterState_type`, to represent the type of the `food` element in `SylvesterState`. If there were another complex type whose name was `SylvesterState_type` in the contract from which the code was generated, Artix would name the class generated to support the `food` element `SylvesterState_type_1`.

Example using nested types

If you had an application using the complex type shown in Example 47 on page 72 your application would include two classes to support it, `TweetyBird` and `SylvesterState`.

Example 49 shows the generated Java class for `tweetyBird`.

Example 49: *TweetyBird Class*

```
//Java
public class TweetyBird
{
  public static final String TARGET_NAMESPACE =
   "http://toonville.org/foodstuffs";

  private boolean caged;
  private int granny_proximity;

  public boolean isCaged()
  {
    return caged;
  }

  public void setCaged(boolean val)
  {
    caged=val;
  }
```

Example 49: *TweetyBird Class*

```
  public int getGranny_proximity()
  {
    return granny_proximity;
  }

  public void setGranny_proximity(int val)
  {
    granny_proximity=val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();

    if (caged != null) {
      buffer.append("caged: "+caged+"\n");
    }
    if (granny_proximity != null) {
      buffer.append("granny_proximity: "+granny_proximity+"\n");
    }
    return buffer.toString();
  }
}
```

The generated class for `sylvesterState`, shown in Example 50, has one element, `food`, that is an instance of `TweetyBird`.

Example 50: *SylvesterState Class*

```
//Java
public class SylvesterState
{
  public static final String TARGET_NAMESPACE =
   "http://toonville.org/cats";

  private int hunger;
  private TweetyBird food;
```

Example 50: *SylvesterState Class*

```
  public int getHunger()
  {
    return hunger;
  }

  public void setHunger(int val)
  {
    hunger=val;
  }

  public TweetyBird getFood()
  {
    return food;
  }

  public void setFood(TweetyBird val)
  {
    food=val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();

    if (caged != null) {
      buffer.append("hunger: "+hunger+"\n");
    }
    if (granny_proximity != null) {
      buffer.append("food: "+food+"\n");
    }
    return buffer.toString();
  }
}
```

When you set the value of `SylvesterState.food`, you must pass a valid `TweetyBird` object to `setFood()`. Also, when you get the value of `SylvesterState.food`, you are returned a `TweetyBird` object which has its own getter and setter methods. Example 51 shows an example of using the nested type `sylvesterState` in Java.

Example 51: *Working with Nested Complex Types*

```
// Java
```

Example 51: *Working with Nested Complex Types*

```
1   SylvesterState hunter = new SylvesterState();
    hunter.setHunger(25);

2   TweetyBird prey = new TweetyBird();
    prey.setCaged(false);
    prey.setGranny_proximity(0);

3   hunter.setFood(pery);

4   System.out.println("The cat is this hungry:
       "+hunter.getHunger());
    System.out.println("The food is caged:
       "+hunter.getFood().isCaged());

5   TweetyBird outPrey = hunter.getFood();
    System.out.println("Granny is this many feet away:
       "+outPrey.getGranny_proximity());
```

The code in Example 51 does the following:

1. Instantiates a new `SylvesterState` object and sets its `hunger` element to 25.
2. Instantiates a new `TweetyBird` object and sets its values.
3. Sets the `food` element on `hunter`.
4. Prints out the value of the `hunger` element and the value of the `food` element's `caged` element.
5. Gets the `food` element, assigns it to `outPrey` then prints out the `granny_proximity` element.

Deriving a Complex Type from a Simple Type

Overview

Artix supports derivation of a complex type from a simple type. A simple type has, by definition, neither sub-elements nor attributes. Hence, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type.

Example 52 shows an example of a complex type, `internationalPrice`, derived by extension from the `xsd:decimal` simple type to include a currency attribute.

Example 52: *Deriving a Complex Type from a Simple Type by Extension*

```
<complexType name="internationalPrice">
    <simpleContent>
        <extension base="xsd:decimal">
            <attribute name="currency" type="xsd:string"/>
        </extension>
    </simpleContent>
</complexType>
```

The `<simpleContent>` tag indicates that the new type does not contain any sub-elements and the `<extension>` element defines the derivation by extension from `xsd:decimal`.

Java mapping

A complex type derived from a simple type is mapped to a Java class. The class will contain an element, `value`, of the simple type from which the complex type is derived. The class will also have a `get_value()` and a `set_value()` method. In addition, the generated class will have an element, and the associated getter and setter methods, for each attribute that extends the simple type.

Example 53 shows the generated Java class representing `internationalPrice` class generated from Example 52.

Example 53: *internationalPrice Java Class*

```
//Java
public class InternationalPrice
{
  public static final String TARGET_NAMESPACE =
   "http://moneyTree.com";


  private String currency;
  private java.math.BigDecimal _value;

  public String getCurrency()
  {
    return currency;
  }

  public void setCurrency(String val)
  {
    currency = val;
  }

  public java.math.BigDecimal get_value()
  {
    return _value;
  }

  public void set_value(java.math.BigDecimal val)
  {
    _value = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();
    if (currency != null) {
      buffer.append("currency: "+currency+"\n");
    }
    if (_value != null) {
      buffer.append("_value: "+_value+"\n");
    }
    return buffer.toString();
  }
}
```

The value of the currency attribute, which is added by extension, can be accessed and modified using the `getCurrency()` and `setCurrency()` methods. The simple type value (that is, the value enclosed between the `<internationalPrice>` and `</internationalPrice>` tags) can be accessed and modified by the `get_value()` and `set_value()` methods.

Occurrence Constraints

Overview

XMLSchema allows you to specify the minimum and the maximum number of times that an element in a complex type can occur. You specify these occurrence constraints on an element by setting the element's `minOccurs` and `maxOccurs` attributes. The `minOccurs` attribute specifies the minimum number of times the element must occur. The `maxOccurs` attribute specifies the upper limit for how many times the element can occur. For example, if an element, `lives`, were to occur at least twice and no more than nine times in a complex type it would be described as shown in Example 54.

Example 54: *Occurrence Constraints Setting*

```
<complexType name="houseCat">
  <all>
    <element name="name" type="xsd:string" />
    <element name="lives" type="xsd:short" minOccurs="2"
             maxOccurs="9" />
  </all>
</complexType>
```

Given the description in Example 54, a valid `houseCat` element would have a single `name` and at least two `lives`. However, a valid `houseCat` element could not have more than nine `lives`.

Note: When a sequence schema contains a *single* element definition and this element defines occurrence constraints, it is treated as an array. See "SOAP Arrays" on page 92.

Mapping to Java

When a complex type contains an element with its `maxOccurs` attribute set to a value greater than one, the element is mapped to an array of the corresponding Java type. Because XMLSchema requires that the `maxOccurs` attribute of an element is set to a value equal to or greater than the value of the element's `minOccurs`, the code generator will generate a warning if the `minOccurs` attribute is set without a `maxOccurs` attribute. So all valid elements with an occurrence constraint will be mapped into an array.

Example

For example, the complex type, `houseCat`, shown in Example 54 will be mapped to the Java class `HouseCat` shown in Example 55.

Example 55: *HouseCat Java Class*

```
// Java
public class HouseCat
{
  private String name;
  private short[] lives;

  public void setName(String val)
  {
    name=val;
  {
  public String getName()
  {
   return name;
  }

  public void setLives(short[] val)
  {
    lives=val;
  {
  public short[] getLives()
  {
   return lives;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();
    if (name != null)
    {
      buffer.append("name: "+name+"\n");
    }
    if (lives != null)
    {
      buffer.append("lives: "+lives+"\n");
    }
    return buffer.toString();
  }
}
```

The generated code does not force you to obey the min and the max occurrence rules from the contract, but your application code should be sure to obey the contract rules. Attempting to send too few or too many occurrences of an element across the wire will create unpredictable results.

Using XMLSchema any Elements

Overview

An XMLSchema any is a special element used to denote that an element's contents are undefined. An element defined using any can contain any XML data. When mapped to Java, an any element is mapped to a `SOAPElement` as called for in the JAX-RPC specification.

Describing an any in the contract

Example 56 shows the syntax for defining an element as an any in an Artix contract.

Example 56: *Syntax of an any*

```
<any [maxOccurs = max] [minOccurs = min]
     [namespace = ((##any | ##other) | List of (anyURI |
   (##targetNamespace | ##local)))]
     [processContents = (lax | skip | strict)] />
```

Table 4 explains the details of the optional attributes.

Table 4: *Attributes for an any*

Attribute	Explanation
`maxOccurs`	Specifies the maximum number of times the element can occur. Default is `1`.
`minOccurs`	Specifies the minimum number of times the element must occur. Default is `1`.

Table 4: *Attributes for an any*

Attribute	Explanation
`namespace`	Specifies how to determine the namespace to use when validating the contents of the any. Valid entries are: **##any(default)** specifies that the contents of the any can be from any namespace. **##other** specifies that the contents of the any can be from any namespace but the target namespace. **list of URIs** specifies that the contents of the any are from one of the listed namespaces in the space delimited list. The list can contain two special values: • `##local` which correspondes to an empty namespace. • `##targetNamespace` which corrensponds to the tager namespace of the schema in which the any is defined.
`processContents`	Specifies how the contents of the any are validated. Valid entries are: **strict(default)** specifies that the contents of the any must be a valid and well-formed XML document. **skip** specifies that no validation is done on the contents of the any. The only constraint is that it must be a well-formed XML element. **lax** specifies that if there is an XMLSchema definition available to validate the contents of the any, then it must be valid. If there is no XMLSchema definition available, then validation is skipped.

Example 57 shows the definition of a type, `wildCard`, that contains an any. The contents of `wildCard` can be defined in any, or no, namespace and the validation of the contents is only performed if there is schema available.

Example 57: *Complex Type with an any*

```
<complexType name="wildCard">
  <sequence>
    <any namespace="##any" processContents="lax" />
  </sequence>
</complexType>
```

Mapping to Java

XMLSchema any elements are mapped to a Java element of type `javax.xml.soap.SOAPElement`. The member is named _any and it is given associated setter and getter methods. If a complex type contains more than one any element the additional any elements are named _any_*n*, where n is an integer starting at one. For example, if a complex type had two any elements the generated Java type would have two `javax.xml.soap.SOAPElement` members, _any and _any_1.

Example 58 shows the Java class generated for the complex type `wildCard`, shown in Example 57 on page 86.

Example 58: *Generated Java Class with an any*

```
// Java
import java.util.*;
import javax.xml.soap.SOAPElement;

public class WildCard
{
  public static final String TARGET_NAMESPACE =
   "http://packageTracking.com/types/packageTypes";

  private javax.xml.soap.SOAPElement _any;

  public javax.xml.soap.SOAPElement get_any()
  {
    return _any;
  }
```

Example 58: *Generated Java Class with an any*

```
  public void set_any(javax.xml.soap.SOAPElement val)
  {
    this._any = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer();
    if (_any != null) {
        buffer.append("_any: "+_any+"\n");
        }
    return buffer.toString();
  }
}
```

If the `minOccurs` or `maxOccurs` attribute of the `any` element are set, then the Java element is mapped to an array of `SOAPElement`. For example, if the `any` element in `wildCard` had `maxOccurs="4"`, the `_any` member of the generated Java class would be a `javax.xml.soap.SOAPElement[]`.

Parsing an any

The fact that an `any` element can hold any well-formed XML data makes it very flexible. However, that flexibility requires that your application is designed to handle all the possible contents of the `any`.

For most applications, the contents of the `any` will have a finite number of forms and these are known at development time. For example, if your application is retrieving student records from a college database it may receive different records based on if the student is a graduate student or an under graduate student. In cases where you know at development time the possible contents of the `any`, you can query the `any` for the name of its root element using `SOAPElement.getElementName()` and determine from the returned `javax.xml.soap.Name` how to process the contents.

Note: Because the contents of the `any` is an XML document made up entirely of text, you do not neccesarily need to determine the form of the data. You can still extract the contents using the `SOAPElement`'s methods.

Example 59 shows code for querying the any in `WildCard` for its element name. Once the element is determined, the application uses the local part of the name to determine how to process the contents of the any.

Example 59: *Determining the Contents of an any*

```
// Java
import java.util.*;
import javax.xml.soap.*;

WildCard dataHolder;

// Client proxy, proxy, instantiated earlier
dataHolder = proxy.getRecord();
SOAPElement studentRec=dataHolder.get_any();

// Get the root element name of the returned record
Name recordType = studentRec.getElementName();

if (recordType.getLocalName().equals("gradRec"))
{
  // process the data as a graduate student record
}
if (recordType.getLocalName().equals("ugradRec"))
{
  // process the data as a graduate student record
}
```

You can parse the XML content of the any using the `SOAPElement.getChildElements()` method. `getChildElements()` returns a Java `Iterator` containing a list of `javax.xml.soap.Node` elements representing the nodes of the XML document contained in the any. These nodes will in turn either be `SOAPElement` nodes or `javax.xml.soap.Text` nodes which will require further parsing.

Example 60 shows code for extracting the data from an any containing a `houseCat`, defined in Example 54 on page 81.

Example 60: *Parsing the Contents of an any*

```
// Java
import java.util.*;
import javax.xml.soap.*;

WildCard dataHolder;
```

Example 60: *Parsing the Contents of an any*

```
1   // Client proxy, proxy, instantiated earlier
    dataHolder = proxy.getCat();
    SOAPElement catHolder = dataHolder.get_any();

2   // Get the XML node from the returned any
    Iterator catIt = catHolder.getChildElements();

3   if (catIt.hasNext())
    {
      System.out.println("The cat's name is
       "+catIt.next().getValue());
    }
    else
    {
      System.out.println("Malformed houseCat: No elements.");
      return(-1);
    }
4   if (catIt.hasNext())
    {
      for (Node index=catIt.next(); (catIt.hasNext());
           index=catIt.next())
      {
        System.out.println("The cat lived
       "+index.getValue()+"years");
      }
    else
    {
      System.out.println("Malformed houseCat: No lives.");
      return(-1);
    }
    }
```

The code in Example 60 does the following:

1. Gets the data and extracts the any from it.
2. Gets the children elements of the any.
3. Checks if there are any children elements. If there are, print the name. If not, print an error message.
4. Checks if there are any more children elements. If there are, iterate through the list and print the lives. If not, print an error message.

To get the value of the nodes, the code uses the `getValue()` method of the node. For a `SOAPElement` node, `getValue()` returns the value of the element if it has one, or it returns the value of the first child element that has one. For example, if the `SOAPElement` contains the element `<name>Joe</name>`, `getValue()` returns `Joe`. If the `SOAPElement` contains `<houseCat><name>Joe</name><lives>12</lives></houseCat>`, `getValue()` returns `Joe`. For a `Text` node, `getValue()` returns the text stored in the node.

Putting content into an any

When adding content into an any, you build up the XML document contained in the any from scratch. The `SOAPElement` provides a number of methods for adding attributes and elements. It has methods for setting the value of the contained elements.

Example 61 shows the code for creating an any element containing the XML document `<houseCat><name>Joe</name><lives>12</lives></houseCat>`.

Example 61: *Building an any*

```
//Java
import java.util.*;
import javax.xml.soap.*;

1  SOAPElementFactory factory = SOAPElementFactory.newInstance();

2  SOAPElement anyContent = factory.create("houseCat");

3  SOAPElement tmp = anyContent.addChildElement("name");
   tmp.addTextNode("Joe");

4  tmp = anyContent.addChildElement("lives");
   tmp.addTextNode("12");

5  WildCard dataHolder = new WildCard();
   dataHolder.set_any();
```

The code in Example 61 does the following:

1. Gets an instance of the `SOAPElementFactory`.
2. Creates a new `SOAPElement`, using the factory, to hold the contents of the any.
3. Adds the `<name>` child element and set its value.
4. Adds the `<lives>` child element and set its value.

5. Creates a new `WildCard` and set the `any` element to the newly created `SOAPElement`.

More information

For a detailed description of the classes used to represent and work with any elements read the *SOAP with Attachments API for Java™ (SAAJ) 1.2* specification.

SOAP Arrays

Overview

SOAP encoded arrays support the definition of multi-dimensional arrays, sparse arrays, and partially transmitted arrays. They are mapped directly to Java arrays of the base type used to define the array.

Syntax of a SOAP Array

SOAP arrays can be described by deriving from the `SOAP-ENC:Array` base type using the `wsdl:arrayType`. The syntax for this is shown in Example 62.

Example 62: *Syntax for a SOAP Array derived using wsdl:arrayType*

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

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

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in Example 63.

Example 63: *Syntax for a SOAP Array derived using an Element*

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

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

Java mapping

SOAP arrays, like basic arrays, are mapped to Java arrays and do not cause a new class to be generated to represent them. Instead, any message part that was specified in the Artix contract as being of type *`ArrayType`* or any element of another complex type that was of type `ArrayType` in the Artix contract would be mapped to an array of the appropriate type.

For example, the SOAP Array, `SOAPStrings`, shown in Example 64 defines a one-dimensional array of strings. The `wsdl:arrayType` attribute specifies the type of the array elements, `xsd:string`, and the number of dimensions, `[]` implying one dimension.

Example 64: *Definition of a SOAP Array*

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

Any message part of type `SOAPStrings` and any complex type element of type `SOAPStrings` would be mapped to `String[]`. So the contract fragment shown in Example 65, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 65: *Operation Using an Array*

```
...
<message name="badLang">
  <part name="statement" type="SOAPStrings" />
</message>
<portType name="censor">
  <operation name="celebWasher">
    <input message="badLang" name="badLang" />
  </operation>
</portType>
...
```

Multi-dimensional arrays

Multi-dimensional arrays are also mapped to a Java array of the appropriate type. In the case of a multi-dimensional array, the generated Java array would have the same dimensions as the SOAP array. For example, if `SOAPStrings` were mapped to a two-dimensional array, as shown in Example 66, the mapping of `celebWasher()` would take a parameter, `badLang`, of type `String[][]`.

Example 66: *Definition of a two-dimensional SOAP Array*

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

Sparse and partially transmitted arrays

Sparse and partially transmitted arrays are simply special cases of how an array is populated. A sparse array is an array where not all of the elements are set. For example, if you had an array, `intArray[]`, of 10 integers and only filled in `intArray[1]`, `intArray[6]`, and `intArray[9]`, it would be considered a sparse array.

A partially transmitted array is an array where only a certain range of elements are set. For example, if you had a two dimensional array, `hotMatrix[`*x*`][`*y*`]`, and only put values in elements where $9 > x > 5$ and $4 > y > 0$, it would be considered a partially transmitted array.

Artix handles both of these cases automatically for you. However, due to differences between Web service implementations, an Artix Java client may receive a fully allocated array with only a few elements containing valid data.

Lists

Overview

XMLSchema supports a mechanism for defining data types that are a list of space separated simple types. An example of an element, `simpleList`, using a list type is shown in Example 67.

Example 67: *List Type Example*

```
<simpleList>apple orange kiwi mango lemon lime<\simpleList>
```

In Java code list types are mapped into arrays.

Defining list types in XMLSchema

XMLSchema list types are simple types and as such are defined using a `<simpleType>` element. The most common syntax used to define a list type is shown in Example 68.

Example 68: *Syntax for List Types*

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value"/>
    <facet value="value"/>
    ...
  </list>
</simpleType>
```

The value given for *`atomicType`* defines the type of the elements in the list. It can only be one of the built in XMLSchema atomic types, like `xsd:int` or `xsd:string`, or a user-defined simple type that is not a list.

In addition to defining the type of elements listed in the list type, you can also use facets to further constrain the properties of the list type. Table 5 shows the facets used by list types.

Table 5: *List Type Facets*

Facet	Effect
`length`	Defines the number of elements in an instance of the list type.

Table 5: *List Type Facets*

Facet	Effect
`minLength`	Defines the minimum number of elements allowed in an instance of the list type.
`maxLength`	Defines the maximum number of elements allowed in an instance of the list type.
`enumeration`	Defines the allowable values for elements in an instance of the list type.
`pattern`	Defines the lexical form of the elements in an instance of the list type. Patterns are defined using regular expressions.

For example, the definition for the `simpleList` element shown in Example 67 on page 95, is shown in Example 69.

Example 69: *Definition for simpleList*

```
<simpleType name="simpleListType">
  <list itemType="string"/>
</simpleType>
<element name="simpleList" type="simpleListType"/>
```

In addition to the syntax shown in Example 68 on page 95 you can also define a list type using the less common syntax shown in Example 70.

Example 70: *Alternate Syntax for List Types*

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
    </simpleType>
  </list>
</simpleType>
```

Mapping of list types in Java

List types are mapped to Java arrays and do not cause a new class to be generated to represent them. Instead, any message part that was specified in the Artix contract as being of type *`listType`* or any element of another complex type that was of type *`listType`* in the Artix contract would be mapped to an array of the type specified by the `itemType` attribute.

For example, the list type, `stringList`, shown in Example 71 defines a list of strings that must have at least two elements and no more than six elements. The `itemType` attribute specifies the type of the list elements, `xsd:string`. The facets `minLength` and `maxLength` set the size constraints on the list.

Example 71: *Definition of stringList*

```
<simpleType name="stringList">
  <list itemType="xsd:string">
    <minLength value="2" />
    <maxLength value="6"/>
  </list>
</simpleType>
```

Any message part of type `stringList` and any complex type element of type `stringList` would be mapped to `String[]`. So the contract fragment shown in Example 72, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 72: *Operation Using a List*

```
...
<message name="badLang">
  <part name="statement" type="stringList" />
</message>
<portType name="censor">
  <operation name="celebWasher">
    <input message="badLang" name="badLang" />
  </operation>
</portType>
...
```

Enumerations

Overview

In XMLSchema, enumerations are described by derivation of a simple type using the syntax shown in Example 73.

Example 73: *Syntax for an Enumeration*

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

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

For example, an XML document with an element defined by the enumeration `widgetSize`, shown in Example 74, would be valid if it were `<widgetSize>big</widgetSize>`, but not if it were `<widgetSize>big,mungo</widgetSize>`.

Example 74: *widgetSize Enumeration*

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

Mapping to a Java class

Artix maps enumerations to a Java class whose name is taken from the schema type's `name` attribute. So Artix would generate a class, `WidgetSize`, to represent the `widgetSize` enumeration.

Note: If the enumeration is an anonymous type nested inside of a complex type, the naming of the generated Java class follows the same pattern as laid out in "Nesting with Anonymous Types" on page 73.

The generated class contains two static public data members for each possible case value. One, *`_caseNValue`*, holds the data value of the enumeration instance. The other, *`caseNValue`*, holds an instance of the class associated with the data value. The generated class also contains four public methods:

fromValue() returns the representative static instance of the class based on the value specified. The specified value must be of the enumeration's type and be a valid value for the enumeration. If an invalid value is specified an exception is thrown.

fromString() returns the representative static instance of the class based on a string value. The value inside the string must be a valid value for the enumeration or an exception will be thrown.

getValue() returns the value for the class instance on which it is called.

toString() returns a stringified representation of the class instance on which it is called.

For example Artix would generate the class, `WidgetSize`, shown in Example 75, to represent the enumeration, `widgetSize`, shown in Example 74 on page 98.

Example 75: *WidgetSize Class*

```
// Java
public class WidgetSize
{
  public static final String TARGET_NAMESPACE =
   "http://widgetVendor.com/types/widgetTypes";
```

Example 75: *WidgetSize Class*

```
private final String _val;

public static final String _big = "big";
public static final WidgetSize big = new WidgetSize(_big);

public static final String _large = "large";
public static final WidgetSize large = new WidgetSize(_large);

public static final String _mungo = "mungo";
public static final WidgetSize mungo = new WidgetSize(_mungo);

public static final String _gargantuan = "gargantuan";
public static final WidgetSize gargantuan = new
 WidgetSize(_gargantuan);

protected WidgetSize(String value)
{
  _val = value;
}

public String getValue()
{
  return _val;
};
```

Example 75: *WidgetSize Class*

```
public static WidgetSize fromValue(String value)
{
  if (value.equals("big"))
  {
    return big;
  }
  if (value.equals("large"))
  {
    return large;
  }
  if (value.equals("mungo"))
  {
    return mungo;
  }
  if (value.equals("gargantuan"))
  {
    return gargantuan;
  }
  throw new IllegalArgumentException("Invalid enumeration
 value: "+value);
  };

public static WidgetSize fromString(String value)
{
  if (value.equals("big"))
  {
    return big;
  }
  if (value.equals("large"))
  {
    return large;
  }
  if (value.equals("mungo"))
  {
    return mungo;
  }
  if (value.equals("gargantuan"))
  {
    return gargantuan;
  }
  throw new IllegalArgumentException("Invalid enumeration
 value: "+value);
  };
```

Example 75: *WidgetSize Class*

```
  public String toString()
  {
    return ""+_val;
  }
}
```

Working with enumerations in Java

Unlike the classes generated to represent complex types, the Java classes generated to represent enumerations do not need to be specifically instantiated, nor do they provide setter methods. Instead, you use the `fromValue()` or `fromString()` methods on the class to get a reference to one of the static members of the enumeration. Once you have the reference to your desired member, you use the `getValue()` method on that member to determine the value for the member.

If you were working with the `widgetSize` enumeration, shown in Example 74 on page 98, to build an ordering system, you would need a way to enter the size of the widget you wanted to order and then store that choice as part of the order. Example 76 shows a simple text entry method for getting the proper member of the enumeration using `fromValue()`,

Example 76: *Using fromValue() to Get a Member of an Enumeration*

```
// Java
temp = new String();
WidgetSize ordered_size;

// Get the type of widgets to order
System.out.println("What size widgets do you want?");
System.out.println("Big");
System.out.println("Large");
System.out.println("Mungo");
System.out.println("Gargantuan");
temp = inputBuffer.readLine();

ordered_size = WidgetSize.fromValue(temp);
```

Because the value used to define the cases of the enumeration is a string, `fromValue()` takes a `String` and returns the member based on the value of the string. In this example, `fromString()` is interchangeable with `fromValue()`. However, if the value of the enumeration were integers, `fromValue()` would take an `int`.

To print the bill you will need to display the size of the widgets ordered. To get the value of the ordered widgets, you could use the `getValue()` method to retrieve the value of the enumeration or you could use the `toString()` method to return the value as a `String`. Example 77 uses `getValue()` to return the value of the enumeration retrieved in Example 76 on page 102

Example 77: *Using getValue()*

```
// Java
String sizeVal = ordered_size.getValue();
System.out.println("You ordered "+sizeVal+" sized widgets.");
```

Deriving Types Using <complexContent>

Overview

Using XMLSchema, you can derive new complex types by extending other complex types using the `<complexContent>` element. When generating the Java class to represent the derived complex type, Artix extends the base type's class. In this way, the Artix-generated Java code preserves the inheritance hierarchy intended in the XMLSchema.

Schema syntax

You derive complex types from other complex types by using the `<complexContent>` element and the `<extension>` element. The `<complexContent>` element specifies that the included data description includes more than one field. The `<extension>` element, which is part of the `<complexContent>` definition, specifies the base type being extended to create the new type. The base type is specified by the `<extension>` element's `base` attribute.

Within the `<extension>` element, you define the additional fields that make up the new type. All elements that are allowed in a complex type description are allowable as part of the new type's definition. For example, you could add an anonymous enumeration to the new type, or you could use the `<choice>` element to specify that only one of the new fields is to be valid at a time.

Example 78 shows an XMLSchema fragment that defines two complex types, `widgetOrderInfo` and `widgetOrderBillInfo`. `widgetOrderBillInfo` is derived by extending `widgetOrderInfo` to include two new fields, `orderNumber` and `amtDue`.

Example 78: *Deriving a Complex Type by Extension*

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:decimal"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:QName" use="optional" />
</complexType>
```

Example 78: *Deriving a Complex Type by Extension*

```
<complexType name="widgetOrderBillInfo">
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:boolean"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Generated Java code

As with all complex types defined in a contract, Artix generates a class to represent complex types derived by extension. When the complex type is derived by extension, the generated class extends the base class generated to support the base complex type in the contract.

For example, the schema in Example 78 on page 104 would result in the generation of two Java classes, `WidgetOrderInfo` and `WidgetBillOrderInfo`. `WidgetOrderBillInfo` would extend `WidgetOrderInfo` because `widgetOrderBillInfo` is derived by extension from `widgetOrderInfo`. Example 79 shows the generated class for `widgetOrderBillInfo`.

Example 79: *WidgetOrderBillInfo*

```
// Java
public class WidgetOrderBillInfo extends WidgetOrderInfo
{
  public static final String TARGET_NAMESPACE =
   "http://widgetVendor.com/types/widgetTypes";

  private boolean amtDue;
  private String orderNumber;

  public boolean isAmtDue()
  {
    return amtDue;
  }
```

Example 79: *WidgetOrderBillInfo*

```
  public void setAmtDue(boolean val)
  {
    this.amtDue = val;
  }

  public String getOrderNumber()
  {
    return orderNumber;
  }

  public void setOrderNumber(String val)
  {
    this.orderNumber = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer(super.toString());
    buffer.append("amtDue: "+amtDue+"\n");
    if (orderNumber != null)
    {
      buffer.append("orderNumber: "+orderNumber+"\n");
    }
    return buffer.toString();
  }
}
```

Holder Classes

Overview

WSDL allows you to describe operations that have multiple output parameters and operations that have in/out parameters. Because Java does not support pass-by-reference, as C++ does, the JAX-RPC 1.1 specification prescribes the use of holder classes as a mechanism to support output and in/out parameters in Java. The holder classes for the Java primitives, and their associated wrapper classes, are packaged in `javax.xml.rpc.holders`. The names of the holder classes start with a capital letter and end with the `Holder` postfix. The name of the holder class for `long` is `LongHolder`. For primitive wrapper classes, `Wrapper` is placed after the class name and before `Holder`. For example, the holder class for `Long` is `LongWrapperHolder`.

For complex types, Artix generates holder classes to represent the complex type when needed. The generated holder classes follows the same naming convention as the primitive holder classes and implement the `javax.xml.rpc.holders.Holder` interface. For example, the holder class for a complex type, `hand`, would be `HandHolder`.

All holder classes provide the following:

- A public field named `value` of the mapped Java type. For example, a `HandHolder` would have a `value` field of type `Hand`.
- A constructor that sets `value` to a default.
- A constructor that sets `value` to the value of the passed in parameter.

Working with holder classes

A holder class is used in the generated Java code when an operation described in your Artix contract either has an output message with multiple parts or when an operation's input message and output message share a part. For a part to be shared it must have the same name and type in both messages. Example 80 shows an example of an operation that would require holder classes in the generated Java code.

Example 80: *Multiple Output Parts*

```
<message name="incomingPackage">
  <part name="ID" type="xsd:long" />
</message>
```

Example 80: *Multiple Output Parts*

```
<message name="outgoingPackage">
  <part name="rerouted" type="xsd:boolean" />
  <part name="destination" type="xsd:string" />
</message>
<portType name="portal">
  <operation name="router">
    <input message="tns:incomingPackage" name="recieved" />
    <output message="tns:outgoingPackage" name="shipped" />
  </operation>
</portType>
```

Artix will use holder classes for the parameters of the Java method generated to implement the operation, `router`, because the output message has multiple parts. Example 81 shows the resulting Java method signature.

Example 81: *Interface Using Holders*

```
//Java
import java.net.*;
import java.rmi.*;

public interface portal extends java.rmi.Remote
{
  public boolean router(long ID,
                        javax.xml.rpc.holders.StringHolder destination)
  throws RemoteException;
}
```

The first part of the `outgoingPackage` message, `rerouted`, is mapped to a boolean return value because it is the first part in the output message. However, the second output message part, `destination`, is mapped to a holder class because it has to be mapped into the method's parameter list.

An example of an application that implements the `portal` interface might be one that determines if a package has reached its final destination. The `router` method would check to see if it need to be forwarded to a new destination and reset the destination appropriately. Example 82 shows how a server might implement the router method.

Example 82: *Portal Implementation*

```
//Java
import java.net.*;
import java.rmi.*;

// The methods boolean belongsHere() and
// String getFinalDestination() are left
// for the reader to implement.

public class portalImpl
{
  public boolean router(long ID,
                javax.xml.rpc.holders.StringHolder destination)
  {
    if(belongsHere(ID))
    {
      return false;
    }

    destination.value = getFinalDestination(ID);
    return true;
  }

}
```

Example 83 shows a client calling `router()` on a portal service.

Example 83: *Client Calling router()*

```
//Java
StringHolder destination = new StringHolder();
long ID = 1232;
boolean continuing;
```

Example 83: *Client Calling router()*

```
// proxy portalClient obtained earlier
continuing = portalClient.router(ID, destination);

if (continuing)
{
  System.out.println("Package "+ID+" is going to
   "+destination.value);
}
```

Using SOAP with Attachments

Overview

When a contract specifies that one or more of an operation's messages are being sent using SOAP with attachments, also called a MIME multi-part related message, Artix treats the data being passed as an attachment differently than it would normaly. The JAX-RPC specification defines specific Java data types to be used when using SOAP attachments. The data mappings for the data passed as a SOAP attachment is derived from the MIME type specified in the contract for the message part.

In addition, Artix support the use of `javax.activation.DataHandler` objects for handling SOAP attachments. `DataHandler` objects provide a generic means of dealing with the data passed as a SOAP attachment. They also allow you to directly access the stream representation of the data sent as a SOAP attachment.

JAX-RPC mappings

When Artix generates code for an operation that has one or more of its message bound to a SOAP with attachment payload format, it inspects the binding to see which parts of the bound message are being sent as attachments. For the message parts that are to be sent as attachments, it disregards the data type mappings described in previous sections and maps the corresponding method parameter based on the MIME type specified for the part in the contract. Table 6 shows the mappings for the supported MIME types.

Table 6: *MIME Type Mappings*

MIME Type	Java Type
`image/gif`[a]	`java.awt.Image`
`image/jpeg`	`java.awt.Image`
`text/plain`	`java.lang.String`
`text/xml`	`javax.xml.transform.Source`
`application/xml`	`javax.xml.transform.Source`
`multipart/*`	`javax.mail.internet.MimeMultipart`

a. Artix only supports the decoding of images in the GIFF format. It does not support the encoding of images into the GIFF format.

For example, the contract shown in Example 84 has one operation, `store`, whose input message has three parts: a patient name, a patient ID number, and a `base64Binary` buffer to hold an image. The input message is bound to a SOAP message with attachments using the `<mime:multiPart>` element.

Example 84: *Using SOAP with Attachments*

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

Example 84: *Using SOAP with Attachments*

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

The binding specifies that only one part of the message, the `base64Binary` buffer, is to be passed as an attachment using the MIME type `image/jpeg`. The other two parts of the message are to be passed in the SOAP body of the message. If the operation were bound to a standard SOAP message, the

generated method would have a `String` paramemter, an `int` parameter, and a `byte[]` parameter. Instead the operation, `store`, is mapped as shown in Example 85.

Example 85: *Java for SOAP with Attachments*

```
// Java
package org.medistor.x_rays;

import java.net.*;
import java.rmi.*;

import java.lang.String;
import java.awt.Image;

public class XRayStorageImpl implements java.rmi.Remote
{
  public boolean store(String patientName,
                       int patientNumber,
                       java.awt.Image xRay) {
        // User code goes in here.
        return false;
  }
}
```

Using DataHandler objects

Artix also provides the option to map SOAP attachments to `javax.activation.DataHandler` objects. To have Artix map SOAP attachments to `DataHandler` objects, use the `-datahandlers` flag when running `wsdltojava`.

When using `DataHander` objects, Artix maps all SOAP attachements to a `DataHandler`, so the contract in Example 84 on page 112 would result in the operation shown in Example 86 as opposed to the one shown in Example 85 on page 114.

Example 86: *SOAP Attachments Using DataHandler Objects*

```
// Java
package org.medistor.x_rays;

import java.net.*;
import java.rmi.*;
```

Example 86: *SOAP Attachments Using DataHandler Objects*

```
import java.lang.String;
import javax.activation.DataHandler;

public class XRayStorageImpl implements java.rmi.Remote
{
  public boolean store(String patientName,
                       int patientNumber,
                       javax.activation.DataHandler xRay)
  {
        // User code goes in here.
        return false;
  }
}
```

Using `DataHandler` objects to manipulate SOAP attachments provides you with greater control over the data being passed in the attachment. As specified in the J2EE specification, `DataHandler` objects have methods that allow you to manipulate the attachment data as either an `Object`, an `InputStream`, or an `OutputStream`. In addition, `DataHandler` objects allow you to querey it for the MIME type for the data being passed in the attachment. For more information on using `DataHandler` objects see the J2EE API documentation at http://java.sun.com/j2ee/1.4/docs/api/index.html.

Note: When creating `DataHandler` objects to be passed in a SOAP attachment, ensure that the MIME type specified in the creator method matches the MIME type specified in the contract.

CHAPTER 5

Creating User-Defined Exceptions

Artix supports the definition of user-defined exceptions using the WSDL <fault> element. When mapped to Java, the <fault> element is mapped to a throwable exception on the associated Java method.

In this chapter

This chapter discusses the following topics:

Describing User-defined Exceptions in an Artix Contract	page 118
How Artix Generates Java User-defined Exceptions	page 120
Working with User-defined Exceptions in Artix Applications	page 122

Describing User-defined Exceptions in an Artix Contract

Overview

Artix allows you to create user-defined exceptions that your service can propagate back to its clients. As with any information that is exchanged between a service and client in Artix, the exception must be described in the Artix contract. Describing a user-defined exception in an Artix contract involves the following:

- Describing the message that the exception will transmit.
- Associating the exception message to a specific operation.
- Describing how the exception message is bound to the payload format used by the service.

This section will deal with the first two tasks involved in describing a user-defined exception. The third task, describing the binding of the exception to a payload format, is beyond the scope of this book. For information on binding messages to specific payload formats in an Artix contract read *Designing Artix Solutions*.

Describing the exception message

Messages to be passed in a user-defined exception are described in the same manner as the messages used as input or output messages for an operation. The message is described using the `<message>` element. There are no restrictions on the data types that can be passed as part of an exception message or on the number of parts the message can contain.

Note: When using SOAP as your payload format, you are restricted to using only a single part in your exception messages.

Example 87 shows a message description in an Artix contract.

Example 87: *Message Description*

```
<message name="notEnoughInventory">
  <part name="numInventory" type="xsd:int" />
</message
```

For more information on describing a message in an Artix contract, read *Designing Artix Solutions*.

Associating the exception with an operation

Once you have described the message that will be transmitted for your user-defined exception, you need to associate it with an operation in the contract. To do this you add a `<fault>` element to the operation's description. A `<fault>` element takes the same attributes as the `<input>` and `<output>` elements. The `message` attribute specifies the `<message>` element describing the data passed by the exception. The `name` attribute specifies the name by which the exception will be referenced in the binding section of the contract.

Example 88 shows an operation description that uses the message described in Example 87 on page 118 as a user-defined exception.

Example 88: *Operation with a User-defined Exception*

```
<operation name="getWidgets">
  <input message="tns:widgetSizeMessage" name="size" />
  <output message="tns:widgetCostMessage" name="cost" />
  <fault message="tns:notEnoughInventory" name="notEnough" />
</operation>
```

The operation described in Example 88, `getWidgets`, takes one argument denoting the size of the widgets to get from inventory and returns a message stating the cost of the widgets. If the operation cannot get enough widgets, it throws an exception, containing the number of available widgets, back to the client.

How Artix Generates Java User-defined Exceptions

Overview

As specified in the JAX-RPC specification, fault messages describing a user-defined exception in an Artix contract are mapped to a Java exception class by the Artix code generator. The generated class extends the Java `Exception` class so that it can be thrown. It will have one private data member of the type specified in the contract's message part to represent each part of the message, a creation method that allows you to specify the values of each data member, and the associated getter and setter methods for each data member. In addition, the generated class will have a `toString()` method.

The naming scheme for the generated exception class follows that for the generated classes to represent a complex type. The name of the class will be taken from the `name` attribute of the exception's message description and will always start with a capital letter.

Example

Example 89 shows the generated exception class for the fault message in Example 87 on page 118.

Example 89: *Generated Java Class*

```
//Java
import java.util.*;

public class NotEnoughInventory extends Exception
{
  public static final String TARGET_NAMESPACE =
   "http://widgetVendor.com/widgetOrderForm";

  private int numInventory;

  public NotEnoughInventory(int numInventory)
  {
    super();
    this.numInventory = numInventory;
  }
```

Example 89: *Generated Java Class*

```
  public int getNumInventory()
  {
    return numInventory;
  }

  public void setNumInventory(int val)
  {
    numInventory = val;
  }

  public String toString()
  {
    StringBuffer buffer = new StringBuffer(super.toString());
    if (size != null)
    {
      buffer.append("numInventory: "+numInventory+"\n");
    }
    return buffer.toString();
  }
}
```

The TARGET_NAMESPACE member of the class is the target namespace specified for the Artix contract. It will be the same for all classes generated from a particular contract.

Working with User-defined Exceptions in Artix Applications

Overview

Because Artix generates a standard Java exception class for user-defined exceptions, they are handled like any non-Artix exception in a Java application. The implementation of the service can instantiate and throw Artix user-defined exceptions if they encounter the need. The client invoking the service, as long as it is a JAX-RPC compliant Java web service client or an Artix C++ client, will catch Artix user-defined exceptions like any other exception and inspect the contents using the standard methods.

Example

Example 90 shows how a server implementing the `getWidgets` operation, shown in Example 88 on page 119, might instantiate and throw a `NotEnoughInventory` exception.

Example 90: *Throwing a User-defined Exception*

```
//Java
...
// checkInventory() is left for the reader to implement
// size and numOrdered are parameters passed into the operation
if (numOrdered > checkInventory(size))
{
  throw NotEnoughInventory(checkInventory(size));
}
```

Example 91 shows how a client might catch and report the exception thrown by the server.

Example 91: *Catching a User-defined Exception*

```
// Java
...
try
{
  long cost = getWidgets(size, numOrdered);
}
```

Example 91: *Catching a User-defined Exception*

```
catch(NotEnoughInventory nei)
{
  // get the value stored in the exception
  int numInventory = nei.getNumInventory();
  System.out.println("The factory only has "+numInventory+
                     " widgets of size "+size+".");
}
```

CHAPTER 6

Working with Artix Type Factories

Artix uses generated type factories to support a number of advanced features including XMLSchema anyType support and message contexts.

In this chapter

This chapter discusses the following topics:

Introduction to Type Factories	page 126
Registering Type Factories	page 128
Getting Type Information From Type Factories	page 131

Introduction to Type Factories

What are type factories?

Artix type factories are generated classes that allow the Artix bus to dynamically create instances of user defined types. They are used to support Artix functionality that manipulate data using generic Java `Object` instances such as working with XMLSchema `anyType` instances, message contexts, and SOAP headers.

Using type factories in your applications

To use type factories in your Artix applications you need to do the following:

1. Generate the type factories for all of the XMLSchema types and XMLSchema elements used by your application.
2. Register the type factories with the bus used by your application.

Once the type factories are registered with the bus, it will use the type factories to create the proper holders for any data that needs them. In addition, you can also use the functions on the type factories to get information about the types used in your application or to dynamically instantiate classes for your data types.

Generating type factories

`wsdltojava` automatically generates a type factory for all user-defined types in a contract when it generates the code for them. The type factory class is named by postfixing `TypeFactory` onto the port type's name. For example if you generated Java code for a port type named `packageDepot`, the generated type factory class would be `packageDepotTypeFactory`.

Additionally, you can pass `wsdltojava` an XMLSchema document that defines types used by your application and it will generate the classes and type factory for the defined types.

Each contract or XMLSchema document results in one type factory that supports all of the types and elements defined by it. The generated type factory will also support all of the types and elements defined by any imported XMLSchema documents. So, if your application only uses the complex types defined in its own contract you will only need to register one type factory. However, if your application uses types defined in a second XMLSchema document, you will need to generate and register the type factory for those types also.

Java packages for anyType support

When using type factories you must import the package `com.iona.webservices.reflect.types.TypeFactory`.

Registering Type Factories

Overview

Before the Artix bus can use the generated type factories, they must be registered with the bus. This is done using the bus' `registerTypeFactory()` method.

Procedure

To register type factories with an application's bus do the following:

1. Get a reference to the application's bus as shown in "Getting a Bus" on page 32.
2. Instantiate the type factories you wish to register with the client proxy as shown in "Instantiating a type factory" on page 128.
3. Register the type factories using `registerTypeFactory()` on the `Bus` object as shown in "Registering a type factory" on page 129.

Instantiating a type factory

The Artix Java code generator automatically generates a type factory for all of the complex types and elements defined in a contract. The type factory class is named by postfixing `TypeFactory` onto the port type's name. For example if you generated Java code for a port type named `packageDepot`, the generated type factory class would be `PackageDepotTypeFactory`.

You instantiate a type factory in the same manner as a typical Java object. Its constructor takes no arguments. Example 92 shows the code to instantiate the type factory for `packageDepot`.

Example 92: *Instantiating a TypeFactory*

```
//Java
PackageDepotTypeFactory factory = new PackageDepotTypeFactory();
```

Registering a type factory

You register a type factory with the bus using its `registerTypeFactory()` method. `registerTypeFactory()` takes an instance of a type factory as its only argument. Example 93 shows code registering a type factory.

Example 93: *Registering a Type Factory*

```
//Java

...
// Bus bus and TypeFactory factory obtained above
bus.registerTypeFactory(factory);
```

To register multiple type factories with the bus, call `registerTypeFactory()` with each additional type factory. Subsequent calls add new type factories to the list of registered type factories.

Determining if type factories are registered

You can get a hash table of the type factories registered with a bus using `getTypeFactoryMap()`. The returned hash table contains the `QName` for the registered type factories and an `ArrayList` of `TypeFactory` objects containing all of the registered type factories. Example 94 shows code for returning the hash table of registered type factories.

Example 94: *Getting Hash Table of Registered Type Factories*

```
//Java
HashMap factMap = bus.getTypeFactoryMap();
```

Example

Example 95 shows an example of registering two type factories, `packageDepotTypeFactory` and `widgetsTypeFactory`.

Example 95: *Registering Type Factories*

```
//Java
import javax.xml.rpc.*;
import com.iona.webservices.reflect.types.*;
...
// Start the bus and create the Artix client proxy
1 Bus bus = Bus.init();
2 packageDepotTypeFactory fact1 = new packageDepotTypeFactory();
widgetsTypeFactory facts = new widgetsTypeFactory();
```

Example 95: *Registering Type Factories*

```
3  bus.registerTypeFactory(fact1);
   bus.registerTypeFactory(fact2);
```

The code in Example 95 does the following:

1. Initializes the bus.
2. Instantiates the type factory that will be registered.
3. Registers the type factories using `registerTypeFactory()`. The first call registers the type factory for the types defined in the `packageDepot` contract. The second call registers the factory for the types defined in the `widgets` contract.

Getting Type Information From Type Factories

Overview

In most cases you will not need to do anything with the type factories once they are registered. The bus automatically handles the creation of type instances for dynamically created data.

However, you can use the type factory's methods to get information about the supported types or dynamically create instances of data types on your own. `TypeFactory` objects have five methods that provide access to the types supported by the factory. They are:

- getSupportedNamespaces()
- getSchemaType()
- getJavaType()
- getJavaTypeForElement()
- getTypeResourceLocation()

getSupportedNamespaces()

`getSupportedNamespaces()` returns an array of strings listing the namespace URIs used in the schema for which the type factory was generated. For example, if your type factory was generated from a contract that contained the fragment shown in Example 96 a calling `getSupportedNamespaces()` on the generated type factory would return an array of strings containing a single entry: `http://packageTracking.com/packageTypes`.

Example 96: *WSDL Fragment*

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

Example 96: *WSDL Fragment*

```
  <types>
    <schema
     targetNamespace="http://packageTracking.com/packageTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <complexType name="packageInfo">
        <sequence>
          <element name="id" type="xsd:string" />
          <any namespace="##any" processContents="lax"
               maxOccurs="4" />
          <element name="size" type="xsd1:packageSize"/>
          <element name="shippingAddress" type="xsd1:Address"/>
        </sequence>
      </complexType>
      ...
    </schema>
  </types>
  ...
  <portType name="packageDepot">
    ...
  </portType>
  ...
</definitions>
```

Example 97 shows code calling `getSupportedNamespaces()`.

Example 97: *getSupportedNamespaces()*

```
//Java

PackageDepotTypeFactory fact = new PackageDepotTypeFactory();
String[] typeNamespaces = fact.getSupportedNamespaces();
```

---

getSchemaType()

`getSchemaType()` returns the QName of the schema type for which the specified class is generated. It takes a `Class` object for a generated type and returns the QName given in the applications contract for the type which resulted in the generated class.

For example, the contract fragment in Example 96 on page 131 would cause a class called `PackageInfo` to be generated to support the XMLSchema complex type `packageInfo`. Calling `getSchemaType()` on an

instance of `packageDepotTypeFactory`, as shown in Example 98, would return a QName whose local part is `packageInfo` and whose namespace URI is `http://packageTracking.com/packageTypes`.

Example 98: *getSchemaType()*

```
// Java
// PackageDepotTypeFactory fact obtained earlier
QName typeName = fact.getSchemaType(PackageInfo.class);
```

---

getJavaType()

`getJavaType()` returns the Java `Class` object generated to support the specified XMLSchema type. It takes the QName of an XMLSchema type defined using a `<type>` element in the contract from which the type factory was generated as an argument. Using the QName, `getJavaType()` finds the `Class` object generated to support the XMLSchema type and returns an instance of it.

For example, the code in Example 99 gets an instance of the generated `PackageInfo` object by passing `getJavaType()` the QName of the `packageInfo` XMLSchema type defined in Example 96 on page 131.

Example 99: *getJavaType()*

```
//Java

1  QName typeName = new
      QName("http://packageTracking.com/packageTypes",
      "packageInfo");
2  // PackageDepotTypeFactory, fact, obtained earlier
   Class typeClass = fact.getJavaType(typeName);
3  PackageInfo newPackage = typeClass.newInstance();
```

The code in Example 99 does the following:

1. Creates the QName for the XMLSchema type.
2. Calls `getJavaType()` on the type factory to get the `Class` object for the XMLSchema type.
3. Uses the returned `Class` object to create a new instance of `PackageInfo`.

getJavaTypeForElement()	`getJavaTypeForElement()` returns the Java `Class` object generated to support the specified XMLSchema element. It takes the QName of an XMLSchema element defined using an `<element>` element in the contract from which the type factory was generated as an argument. Using the QName, `getJavaTypeForElement()` finds the `Class` object generated to support the XMLSchema element and returns an instance of it.
getTypeResourceLocation()	`getTypeResourceLocation()` returns a string containing the location of the contract, or XMLSchema document, for which the type factory was generated.

CHAPTER 7

Working with XMLSchema anyTypes

The XMLSchema anyType allows you to place a value of any valid XMLSchema primitive or named complex type into a message. This flexibility, however, adds some complexity to your applications.

In this chapter

This chapter discusses the following topics:

Introduction to Working with XMLSchema anyTypes	page 136
Setting anyType Values	page 138
Retrieving Data from anyTypes	page 140

Introduction to Working with XMLSchema anyTypes

XMLSchema anyType

The XMLSchema anyType is the root type for all XMLSchema types. All of the primitives are derivatives of this type as are all user defined complex types. As a result, elements defined as being anyType can contain data in the form of any of the XMLSchema primitives as well as any complex type defined in a schema document.

Artix and anyType

In Artix, an anyType can assume the value of any complex type defined within the `<types>` section of an Artix contract. An anyType can also assume the value of any XMLSchema primitive. For example, if your contract defines the complex types `joeFriday`, `samSpade`, and `mikeHammer`, an anyType used as a message part in an operation can assume the value of an element of type `samSpade` or an element of type `xsd:int`. However, it could not assume the value of an element of type `aceVentura` because `aceVentura` was not defined in the contract.

Artix binding support

Artix supports the use of messages containing parts of anyType using payload formats that have a corresponding native construct such as the CORBA any. Currently Artix allows using anyType with the following payload formats:

- SOAP
- Pure XML
- CORBA

Using anyType in Java

When working with interfaces that use anyType parts in it messages, you need to do a few extra things in developing your application. First, you must register the generated type factory classes with the application's bus. See "Registering Type Factories" on page 128.

When using data stored in an anyType, you can also query the object to determine its actual type before inspecting the data. Retrieving data from an anyType is discussed in "Retrieving Data from anyTypes" on page 140.

Java packages for anyType support

When using anyType data and the type factories you must import the following:

- `com.iona.webservices.reflect.types.AnyType`
- `com.iona.webservices.reflect.types.TypeFactory`

Setting anyType Values

Overview

In Artix Java `xsd:anyType` is mapped to `com.iona.webservices.reflect.types.AnyType`. This class provides a number of methods for setting the value of an `AnyType` object. There are setter methods for each of the supported primitive types. In addition, there is an overloaded setter method for storing complex types in an `AnyType`. This method allows you to specify the `QName` for the schema type definition of the content along with the data or you can simply supply the data and Artix will attempt to determine the data's schema type when the object is transmitted.

Setting primitive data

The Artix `AnyType` class provides methods for storing primitive data in an anyType. The setter methods for the primitive types are listed in Table 7. These methods automatically set the data type identifier to the appropriate schema type when they store the data.

Table 7: *anyType Setter Methods for Primitive Types*

Method	Java Type	XMLSchema Type
`setBoolean()`	`boolean`	`boolean`
`setByte()`	`byte`	`byte`
`setShort()`	`short`	`short`
`setInt()`	`int`	`int`
`setLong()`	`long`	`long`
`setFloat()`	`float`	`float`
`setDouble()`	`double`	`double`
`setString()`	`string`	`string`
`setShort()`	`short`	`short`
`setUByte()`	`short`	`ubyte`
`setUShort()`	`int`	`ushort`

Table 7: *anyType Setter Methods for Primitive Types*

Method	Java Type	XMLSchema Type
`setUInt()`	`long`	`uint`
`setULong()`	`BigInteger`	`ulong`
`setDecimal()`	`BigDecimal`	`decimal`

Setting complex type data

You set complex data into any `AnyType` using the `setType()` method. `setType()` can be used in one of two ways. The first is to provide the `QName` of the XMLSchema type describing the data to store in the `AnyType` along with the data. Using this method makes it easier to query the contents of `anyType` objects that were created in the current application space because Artix does not set the type identifier until after it sends the `anyType` across the wire. Example 100 shows code for storing a `widgetSize` in an `anyType`.

Example 100: *Storing Complex Data and Specifying its Type*

```
//Java
widgetSize size = widgetSize.big;
QName qn = new QName("http://widgetVendor.com/types/",
                     "widgetSize");
AnyType aT =new AnyType();
aT.setType(qn, size);
```

The other way is to simply provide the data value to store in the `AnyType` and Artix will determine the XMLSchema type describing the data. From the receiving end this method for storing data in an `anyType` is equivalent to the first method because Artix identifies the contents schema type when it transmits the data. However, the application that store the value will have no way to determine the data type once the value is stored until it is used as part of a remote invocation. Example 101 shows code for storing a `widgetSize` in an `anyType` without providing its `QName`.

Example 101: *Storing Complex Data without a QName*

```
// Java
widgetSize size = widgetSize.big;
AnyType aT =new AnyType();
aT.setType(size);
```

Retrieving Data from anyTypes

Overview

Because an anyType can assume the values of a number of different data types, it is beneficial to be able to determine the type of the data stored in an anyType before trying to use it. If you knew the value's type you could cast the value into the proper Java type and work with it using standard Java methods.

Artix's Java implementation of anyType provides a mechanism for querying the object to determine the schema type of its value. The type identifier is either set when the value is stored in the anyType or if the type is not specified when the value is set, Artix sets it when the data is transported through the bus.

You can also use the standard Java `getClass()` method on the Java `Object` returned from `AnyType.getObject()` to get the Java class of the data stored in the anyType.

Determining the type of an anyType

The Artix Java `AnyType` provides a method, `getSchemaTypeName()`, that returns the `QName` of the schema type of the data stored in the anyType. Example 102 gets the schema type of an anyType and prints it out to the console.

Example 102: *Using getSchemaTypeName()*

```
// Java
import com.iona.webservices.relect.types.*;

AnyType blackBox;

// Client proxy, proxy, instantiated previously
blackBox = proxy.newBox();
QName schemaType = blackBox.getSchemaTypeName();
System.out.println("The type for blackBox is defined in "
                   +schemaType.getNamespaceURI());
System.out.println("blackBox is of type: "
                   +schemaType.getLocalPart());
```

The data stored in an Artix `AnyType` is a stored as a standard Java `Object`, so when the data is extracted you can use the standard `getClass()` method on the returned `Object` to determine its Java type.

Extracting primitive types from an anyType

The Artix `AnyType` provides specific methods for extracting primitive types. Table 8 lists the getter methods for the supported primitive types and the local part of the schema type name returned by `getSchemaType()`. All of the primitive types have `http://www.w3.org/2001/XMLSchema` as their namespace URI.

Table 8: *Methods for Extracting Primitives from AnyType*

Method	Java Type	Schema Type Name
`getBoolean()`	`boolean`	`boolean`
`getByte()`	`byte`	`byte`
`getShort()`	`short`	`short`
`getInt()`	`int`	`int`
`getLong()`	`long`	`long`
`getFloat()`	`float`	`float`
`getDouble()`	`double`	`double`
`getString()`	`String`	`string`
`getUByte()`	`short`	`unsignedByte`
`getUShort()`	`int`	`unsignedShort`
`getUInt()`	`long`	`unsignedInt`
`getULong()`	`BigInteger`	`unsignedLong`
`getDecimal()`	`BigDecimal`	`decimal`

Extracting complex data from an anyType

The Artix AnyType provides a generic method, getType(), that can be used to extract complex data. getType() returns the data stored in the anyType as a Java Object that you can then cast to the proper Java type. Example 103 shows an example of retrieving a widgetSize from an anyType.

Example 103: *Extracting a Complex Type from an anyType*

```
// Java
AnyType any;

// Client proxy, proxy, instantiated earlier
any = proxy.returnWidget();
widgetSize size = (widgetSize)any.getObject();
```

Example

If you had an application that processed orders for computers. It may be that your ordering system could receive orders for laptops and desktops. Because the laptops and desktops are configured differently you've decided that the orders will be sent using anyType elements that the client then processes. You defined the types, laptopOrder and desktopOrder, in the namespace http://myAssemblyLine.com/systemTypes. Example 104 shows code for receiving the order from the server, querying the returned AnyType to see what type of order it is, and then extracting the order from the AnyType.

Example 104: *Working with anyTypes*

```
// Java
import javax.xml.namespace.QName;
import com.iona.webservices.reflect.types.*;

AnyType anyOrder;

// Client proxy, proxy, instantiated earlier
anyOrder = proxy.getSystemOrder();

// Get the schema type of the returned order
QName orderType = anyOrder.getSchemaType();
```

Example 104:*Working with anyTypes*

```
3  if (!(orderType.getNamespaceURI().equals(
         "http://myAssemblyLine.com/systemTypes"))
   {
     // handle the fact that the schema type is from the wrong
     // namespace.
   }

4  if (orderType.getLocalPart().equals("laptopOrder"))
   {
     LapTopOrder order = (LapTopOrder)anyOrder.getType();
     buildLaptop(order);
   }
5  if (orderType.getLocalPart().equals("desktopOrder"))
   {
     DeskTopOrder order = (DeskTopOrder)anyOrder.getType();
     buildDesktop(order);
   }
```

The code in Example 104 on page 142 does the following:

1. Populates `anyOrder`.
2. Queries `anyOrder` for its schema type information.
3. Checks the namespace of the returned type to ensure it correct.
4. Checks if `anyOrder` is a `laptopOrder`. If so, cast `anyOrder` into a `laptopOrder`.
5. Checks if `anyOrder` is a `desktopOrder`. If so, cast `anyOrder` into a `desktopOrder`.

CHAPTER 8

Artix References

An Artix reference is a handle to a particular Artix service instance. Because they can be passed as message parts, Artix references provide a convenient and flexible way of identifying and locating specific services.

In this chapter

This chapter discusses the following topics:

Introduction to Working with References	page 146
Using References in a Factory Pattern	page 154
Using References to Implement Callbacks	page 168

Introduction to Working with References

Overview

An *Artix Reference* is a Java object that fully describes a running Artix service. Artix references have the following features:

- They are a built-in Artix data type.
- They can be passed as a parameter of an operation.
- They can be used to create service proxies for a service described by a particular reference.
- They are the building blocks for the Artix locator and session manager.
- They are transport neutral. An Artix reference can be used to represent any Artix service.

In this section

This section discusses the following topics:

Reference Basic Concepts	page 147
Creating References	page 151
Instantiating Service Proxies Using a Reference	page 153

Reference Basic Concepts

Overview

An Artix reference is a Java object, derived from an XMLSchema definition shipped with Artix, that fully describes a running Artix service. It lists the service's name, the service's contact information, and the service's WSDL location. The data contained in the reference provides an Artix client process with the information needed to instantiate a service proxy to contact the referenced service.

Using references provides you with the ability to generate servants on the fly and pass a client a reference to the newly instantiated servant. It also provides you the ability to write applications that require using a callback mechanism. In addition, the Artix locator and the Artix session manager use references to supply applications with pointers to the services which they are looking-up.

Contents of an Artix reference

An Artix reference encapsulates the following data:

- *Service QName*—the QName of the service with which the reference is associated. The is the name of the service given in the contract defining the service.
- *WSDL location URL*—the location of the service's contract. The WSDL location URL in a reference services two distinct purposes:
  - Service identification—the service is uniquely identified by the combination a WSDL contract and a service QName.
  - WSDL back-up—the reference is fully self-describing.

  Note: If you have loaded the WSDL publishing plug-in, `wsdl_publish`, on the server, the WSDL location URL will point to a dynamically updated copy of the service's contract. See "Accessing WSDL from a reference" on page 148

- *List of ports*—an unbounded sequence of port elements, each of which contains the following data:
  - *Port name*—the name given the port in the contract.
  - *Binding QName*—the qualified name of the binding with which the port is associated.

- *Properties*—a list of opaque properties, which makes the port element arbitrarily extensible. The properties list is typically used to hold transport-specific data and qualities of service. For example, if the port uses CORBA the properties would include the `<corba:policy>` elements used in the WSDL.

The schema definition of a reference

Like all types in Artix, the reference is defined in XMLSchema. The XMLSchema defining a reference is located in the `schema` folder of your Artix Installation and is called `references.xsd`. It can also be found on-line at http://schemas.iona.com/references/references.xsd.

You will need to import the reference schema into the contract of any application that uses references. It is required for Artix to properly generate the Java code for operations using a reference as a parameter and for the bus to properly marshal and unmarshal references passed between endpoints.

Java mapping of a reference

In Java an Artix reference is mapped to a class called `com.iona.schemas.references.Reference`. This class is provided in the libraries shipped with Artix. Your applications that use Artix references will need to import this class.

Accessing WSDL from a reference

An Artix reference contains a pointer to the contract defining the logical service associated with the reference. By default, the reference's WSDL pointer points to the server's local copy of the service contract. However, if the server process is configured to load the WSDL publishing plugin, the reference's WSDL pointer points to an HTTP port from which a client can download a live copy of the service's contract.

Using the default provides a smaller footprint for your server process and does not require opening an additional HTTP port, but it has two main drawbacks:

- Artix needs to be able to read the WSDL in order to instantiate a service proxy for the referenced service and often the client will not have access to the service's local file system.
- The `<port>` definition for the service may not be complete because the service dynamically sets its port attributes at runtime. In particular, a transient servant's on-disk `<port>` definition is always incomplete.

Configuring your servers to load the WSDL publishing plugin avoids these drawbacks. The WSDL publishing plugin provides a continually updated version of a service's in-memory WSDL contract using an HTTP port. Because the WSDL model is always updated, the reference will always point to a complete contract with valid contact information for the service. Also, because the WSDL is published over an available HTTP port, a client always has access to the WSDL when it attempts to instantiate a service proxy.

For information on configuring a service to load the WSDL publish plugin see *Deploying and Managing Artix Solutions*.

References and the Artix router

When references are passed through the Artix router, the router creates a service proxy for each reference. In this way it ensures that messages are correctly delivered to the referenced service. However, this creates two issues that must be considered:

Misconnected Proxies

Because transient servants are not associated with a fixed service, the router must guess at which WSDL service was used as the service template to create the servant. It chooses the first compatible WSDL service it encounters in the router's contract. A compatible WSDL service is a service that uses the same `<portType>` as the service template used to create the transient servant.

If your contract contains a static WSDL service definition and a service template that both use the same `<portType>`, the router will use the first one listed in the contract. If the static service is first, the router will create a proxy that connects to the servant defined by that service and not the transient service that is referenced. The result will be that all messages directed to the transient servant will be silently forwarded to the wrong servant.

To avoid this situation place all service templates in your router's contract before the static WSDL services. This will ensure that the router will select the service template and create a proxy for the transient servant.

Router bloat

Because the router cannot know when a proxy is no longer needed, it reaps any of the proxies it creates. Because of this, a router that handles a large number of references may get quite bloated. To solve this problem Artix

includes a life-cycle service that allows you to configure a reaping schedule for the router. For more information on using the life-cycle service see *Deploying and Managing Artix Solutions*.

Creating References

Overview

References are created by a bus using the `createReference()` method. Before a bus instance can create a reference for a service, the servant implementing the service must be registered with the bus. The process for creating a reference for a service involves three steps:

1. Get a handle to a bus as shown in "Getting a Bus" on page 32.
2. Register the servant with the bus.
3. Create a reference using the service's `QName`.

Registering a servant

Registering a service with the bus is a two step process. The first step is to create an Artix `Servant` instance for your service. Example 105 shows an example of creating a `Servant` for the `WidgetLoader` service. The `Servant` contsructor requires the path of the contract defining the service, an instance of the service's implementation class, and a bus instance.

Example 105:*Creating a ServerFactoryBase*

```
//Java
Servant servant =
  new SingleInstanceServant("./Widgets.wsdl",
                            new WidgetLoaderImpl(), bus);
```

The second step in registering a service with the bus is to register the servant with a bus instance. Servants can be registered as either static or transient. A static servant is registered using `Bus.registerServant()` and has a fixed port address that is defined in its contract. A transient servant is registered using `Bus.registerTransientServant()`. A transient servant is a clone of the service defined in the contract and each servant for a given service will have a unique port number.

For a detailed discussion of registering servants, read "Servant Registration" on page 26.

Creating the reference

Once you have registered a service with the bus, you can create a reference for it using the `QName` returned from the servant registration method. References are created using the bus' `createReference()` method. Example 106 shows the signature for `createReference()`.

Example 106: *createReference()*

```
//Java
Reference createReference(QName service);
```

The method takes in the `QName` of a registered service. The `QName` of a registered service is returned when you register the servant with the bus. Keeping track of the registered service's `QName` when using references is particularly important when working with transient servants. Because they are clones of a service, each instance of a service registered with a transient servant will have a unique `QName` that is generated by the bus.

Example

Example 107 shows the code for generating a reference for a static instance of the `Cling` service.

Example 107: *Creating a Reference*

```
//Java
import com.iona.jbus.*
import com.iona.schemas.references.Reference;

// Initialize a default bus
Bus bus = Bus.init();

// Register the servant
QName name = new QName("http://www.static.com/Cling",
                       "ClingService");
Servant servant = new SingleInstanceServant(new ClingImpl(),
                                            "./cling.wsdl",
                                            bus);
QName clingName = bus.registerServant(servant, name,
                                      "ClingPort");

// Generate the reference for the register Cling Service
Reference clingRef = bus.createReference(clingName);
```

Instantiating Service Proxies Using a Reference

Overview

One of the primary uses of a reference is to create a service proxy for connecting to the referenced service. The bus provides a method, `createClient()`, that takes a reference and returns a JAX-RPC style dynamic proxy for the referenced service.

Creating a service

To create a service proxy from a reference, you need three things:

- a bus
- a reference
- the Java `Class` representing the service's interface

You create service proxy from a reference by calling `createClient()` on the servant's default bus. `createClient()` takes a reference to a service and the service's interface `Class` as parameters. If the call is successful, it returns a JAX-RPC style dynamic proxy for the service referenced. `createClient()`'s signature is shown in Example 108.

Example 108: *Bus.createClient()*

```
Remote Bus.createClient(Reference ref,
                        Class interfaceClass)
throws BusException
```

Example

Example 109 shows the code for creating a service proxy for the Cling service from a reference.

Example 109: *Getting a Bus Reference Inside a Servant*

```
// Java
com.iona.jbus.Bus bus = DispatchLocals.getCurrentBus();

// Reference clingRef obtained earlier
Cling clingProxy = bus.createClient(clingRef, Cling.class);
```

Using References in a Factory Pattern

Overview

A common pattern for working with references is a factory pattern where one object, a factory, creates references for other objects. For example, you could develop a banking service that is responsible for creating and managing accounts. It may have one operation, `get_account`, that returns references to account objects that handle the more low level operations for depositing or withdrawing money from an account. In this instance, your bank implementation object is a factory for account objects.

This section discusses how such a banking service could be developed. The examples used are loosely based on the transient servant demo supplied with Artix. It is located in the `demos/servant_management/transient_sevants` folder of your Artix installation.

In this section

The following topics are discussed in this section:

Bank Service Contract	page 155
Bank Service Implementation	page 162
Bank Service Client	page 165

Bank Service Contract

Overview

The WSDL contract defining the Bank service has several key elements that are required for defining a service that uses references in a factory pattern. The first thing to notice is that the contract imports the XMLSchema definition for Artix references. Also, it defines two interfaces: `Bank` and `Account`. `Bank` defines an operation for returning references to an `Account`. Also, both interfaces have fully described bindings and service definitions.

For detailed information about Artix contracts read *Designing Artix Solutions*.

Importing the reference schema

Any Artix service that uses references needs to include the XMLSchema definition for an Artix reference in its contract. This can be dome in one of two ways. The most common way is to use an `<import>` element to import the XMLSchema definition that is provided with Artix. Example 110 shows a WSDL fragment that imports the reference schema.

Example 110:*Importing the Reference Schema*

```
<import namespace="http://schemas.iona.com/references"
        location="/usr/local/artix/schema/references.xsd" />
```

The other way is to add the reference definition directly into the contract. This is the method shown in the supplied transient servant demo.

You will also need to add an alias for the references namespace to the definitions tag at the top of the contract as shown in Example 111.

Example 111:*Reference Alias*

```
xmlns:reference="http://schemas.iona.com/references"
```

Messages with references

The Bank interface's get_account operation returns a reference to an Account. The message definition for the response of these operations have one part, return, that is of type reference:Reference. Example 112 shows the definition for a message that contains a reference.

Example 112: *Message with a Reference*

```
<message name="bankResponse">
  <part name="return" type="reference:Reference" />
</message>
```

Bank interface

The <portType> defining the Bank interface defines a single operation named get_account. This operation takes a string as input and returns a reference. Example 113 shows the <portType> for the Bank interface.

Example 113: *Bank <portType>*

```
<portType name="Bank">
  <operation name="get_account">
    <input name="acctName" message="tns:accountName"/>
    <output name="return" message="tns:bankResponse"/>
  </operation>
</portType>
```

Account interface

The contract defining the service will also need to include a definition for the Account interface. This interface can either be defined in a separate WSDL fragment that is imported or it can be defined in the same contract as the Bank interface. The transient servant demo defines the Account interface in the same contract.

Bank binding

While an Artix reference can describe a service that uses any of the bindings supported by Artix, they can only be sent using the SOAP binding or the CORBA binding. When using the SOAP binding, you do not need to anything special to send an Artix reference. The transient servant demo supplied with Artix uses a SOAP binding.

The CORBA binding maps an Artix reference into a generic CORBA `Object`. You can do some additional work to create typed CORBA references. For details on how Artix references are mapped into a CORBA binding see the CORBA appendix of *Designing Artix Solutions*.

Account binding

You will also need to add a binding for the referenced service, which in this case is the `Account` interface. The binding for the referenced service can be any one of the supported Artix bindings. The transient servant demo supplied with Artix uses a SOAP binding for the `Account` interface.

Transport definitions

References can be sent over any transport that supports SOAP or CORBA messages. However, because in this example the servants used to service Account objects will be transient, the Account service must use either HTTP or CORBA.

Complete bank contract

Example 114 shows the complete contract used for the code generated in the following discussions about the factory pattern.

Example 114: *Bank Service Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/bus/demos/bank"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.iona.com/transports/http"
  xmlns:references="http://schemas.iona.com/references"
  xmlns:bank="http://www.iona.com/bus/demos/bank"
  targetNamespace="http://www.iona.com/bus/demos/bank"
  name="BankService">
  <import namespace="http://schemas.iona.com/references"
          location="/usr/local/artix/schema/references.xsd" />
```

Example 114: *Bank Service Contract*

```
<message name="accountName">
  <part name="account_name" type="xsd:string"/>
</message>
<message name="bankResponse">
  <part name="return" type="references:Reference"/>
</message>
<message name="get_balance"/>
<message name="get_balanceResponse">
  <part name="balance" type="xsd:float"/>
</message>
<message name="deposit">
  <part name="addition" type="xsd:float"/>
</message>
<message name="depositResponse"/>
<portType name="Bank">
  <operation name="get_account">
    <input name="acctName" message="tns:accountName"/>
    <output name="return" message="tns:bankResponse"/>
  </operation>
</portType>
<portType name="Account">
  <operation name="get_balance">
    <input name="get_balance" message="tns:get_balance"/>
    <output name="get_balanceResponse" message="tns:get_balanceResponse"/>
  </operation>
  <operation name="deposit">
    <input name="deposit" message="tns:deposit"/>
    <output name="depositResponse" message="tns:depositResponse"/>
  </operation>
</portType>
<binding name="BankBinding" type="tns:Bank">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="get_account">
    <soap:operation soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
    <input>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
  </operation>
</binding>
```

Example 114:*Bank Service Contract*

```
<message name="accountName">
  <part name="account_name" type="xsd:string"/>
</message>
<message name="bankResponse">
  <part name="return" type="references:Reference"/>
</message>
<message name="get_balance"/>
<message name="get_balanceResponse">
  <part name="balance" type="xsd:float"/>
</message>
<message name="deposit">
  <part name="addition" type="xsd:float"/>
</message>
<message name="depositResponse"/>
<portType name="Bank">
  <operation name="get_account">
    <input name="acctName" message="tns:accountName"/>
    <output name="return" message="tns:bankResponse"/>
  </operation>
</portType>
<portType name="Account">
  <operation name="get_balance">
    <input name="get_balance" message="tns:get_balance"/>
    <output name="get_balanceResponse" message="tns:get_balanceResponse"/>
  </operation>
  <operation name="deposit">
    <input name="deposit" message="tns:deposit"/>
    <output name="depositResponse" message="tns:depositResponse"/>
  </operation>
</portType>
<binding name="BankBinding" type="tns:Bank">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="get_account">
    <soap:operation soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
    <input>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
  </operation>
</binding>
```

Example 114: *Bank Service Contract*

```
<message name="accountName">
  <part name="account_name" type="xsd:string"/>
</message>
<message name="bankResponse">
  <part name="return" type="references:Reference"/>
</message>
<message name="get_balance"/>
<message name="get_balanceResponse">
  <part name="balance" type="xsd:float"/>
</message>
<message name="deposit">
  <part name="addition" type="xsd:float"/>
</message>
<message name="depositResponse"/>
<portType name="Bank">
  <operation name="get_account">
    <input name="acctName" message="tns:accountName"/>
    <output name="return" message="tns:bankResponse"/>
  </operation>
</portType>
<portType name="Account">
  <operation name="get_balance">
    <input name="get_balance" message="tns:get_balance"/>
    <output name="get_balanceResponse" message="tns:get_balanceResponse"/>
  </operation>
  <operation name="deposit">
    <input name="deposit" message="tns:deposit"/>
    <output name="depositResponse" message="tns:depositResponse"/>
  </operation>
</portType>
<binding name="BankBinding" type="tns:Bank">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="get_account">
    <soap:operation soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
    <input>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/bank"/>
    </input>
    <output>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/bank"/>
    </output>
  </operation>
</binding>
```

Example 114: *Bank Service Contract*

```
  <binding name="AccountBinding" type="tns:Account">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="get_balance">
      <soap:operation soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
      <input>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/bank"/>
      </input>
      <output>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/bank"/>
      </output>
    </operation>
    <operation name="deposit">
      <soap:operation soapAction="http://www.iona.com/bus/demos/bank" style="rpc"/>
      <input>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/bank"/>
      </input>
      <output>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/bank"/>
      </output>
    </operation>
  </binding>
  <service name="BankService">
    <port name="BankPort" binding="tns:BankBinding">
      <soap:address location="http://localhost:0/BankService/BankPort/"/>
    </port>
  </service>
  <service name="AccountService">
    <port name="AccountPort" binding="tns:AccountBinding">
      <soap:address location="http://localhost:0" />
    </port>
  </service>
</definitions>
```

Bank Service Implementation

Overview

The bank service is the factory for accounts in this example. Its operation, `get_account`, returns references to account objects. `get_account` create accounts and registers them as transient servants. The accounts are registered as transient servants to ensure that each new account has a unique port definition and unique reference.

The bank service implementation object

The Bank service defined in the contract will generated an implementation object called `BankImpl`. This object will contain one method, `get_account()`, for which you will provide the logic. In addition, for this example, `BankImpl` has a global data member, `accounts`, that stores a table of the created accounts by their account name. The line declaring `accounts` is in bold because you need to add it to the generated file.

Example 115 shows the generated `BamkImpl` with `accounts` added.

Example 115: *BankImpl*

```
package com.iona.bus.demos.bank;

import java.net.*;
import java.rmi.*;

import java.lang.String;
import com.iona.schemas.references.Reference;

/**
 * com.iona.bus.demos.bank.BankImpl
 */
public class BankImpl implements java.rmi.Remote
{

  Hashtable accounts = new Hashtable();
```

Example 115: *BankImpl*

```
    /**
     * get_account
     *
     * @param: account_name (String)
     * @return: com.iona.schemas.references.Reference
     */
    public com.iona.schemas.references.Reference
   get_account(String account_name) {
        // User code goes in here.
        return new com.iona.schemas.references.Reference();
    }
}
```

---

get_account

The `get_account` operation in the contract is mapped to the `get_account()` method in the bank service's implementation object. `get_account()` first checks the table of accounts to see if one with the given name already exists. If one does exist, it returns the reference to that account. If no account with that name exists, it creates a new `AccountImpl` object and registers it as a transient servant with the bus.

The `AccountImpl` object is registered as a transient servant because transient servants are guaranteed to have a unique port definition in their in-memory contract and that the reference created for each `AccountImpl` object will point to the correct `AccountImpl`. When using static servants, all references point to a single instance of the servant object.

Note: When working with transient servants, you should ensure that the WSDL publishing plug-in is loaded into the server process.

Once the `AccountImpl` object is registered with the bus, `get_account()` generates a reference for the new servant using `bus.createReference()`. This is the reference that is returned to the client. Using the returned reference, the client will create a service proxy to access the new `Account` object.

Example 116 shows the fully implemented `get_account()`.

Example 116: *get_account()*

```
public Reference get_account(String account_name)
{
```

Example 116:*get_account()*

```
1     Reference ref = (Reference)accounts.get(account_name)

2     if (ref == null)
      {
3       AccountImpl acct = new AccountImpl();

4       com.iona.jbus.Bus bus = DispatchLocals.getCurrentBus();

5       String contract = new String("./bank.wsdl");
        Servant servant = new SingleInstanceServant(acct, contract,
                                                    bus);

6       QName name = new QName("http://www.iona.com/bus/demos/bank",
                               "AccountService");
        bus.registerTransientServant(servant, name);

7       ref = bus.createReference(name);

8      accounts.put(account_name, ref);
      }

8     return ref;
}
```

The code in Example 116 does the following:

1. Looks up the account name in the table of existing accounts.
2. Checks to see if an account was found. If a valid account was found skip to step **9**. If not, continue.
3. Creates a new `AccountImpl` for a new account.
4. Gets the bus for this bank servant.
5. Creates a new Artix Servant for the new account.
6. Registers the new Servant as a transient servant with the bus.
7. Creates a reference for the newly registered transient servant.
8. Adds the new reference and account name to the table of accounts.
9. Returns the reference to the client.

Bank Service Client

Overview

The client for the bank service requests accounts and then performs operations on the returned accounts. In this case, the returned accounts are also services implemented by remote Artix servants. Therefore, before the client can invoke operations on the returned accounts, it must create service proxies for them.

Requirements for building the client

While Artix references are fully self-describing, your client code will still require the generated interface for the `Account` service. This interface will be generated into a file called `Account.java` by `wsdltojava`.

Locating the Account service's contract

Artix references contain a pointer to the contract for the referred service. As discussed in "Accessing WSDL from a reference" on page 148, the WSDL pointer in a reference can either point to the server process' local copy of the service contract or, if the WSDL publishing plugin is loaded, to an HTTP port where the in-memory copy of the contract can be obtained.

Because the Bank service registers the Accounts as transient servants, the server's local copy of the contract will not have a valid `<port>` definition any of the Accounts. Therefore, you will need to ensure that the server process has loaded the WSDL publishing plugin.

Client tasks

The client main in this example does four things:

1. Creates a service proxy for the Bank.
2. Invokes `get_account()` on the Bank proxy.
3. Creates a service proxy for an Account using the returned reference.
4. Invokes operations in the Account proxy.

The first two things that the client does are typical Artix client programming steps. Any Artix client will instantiate a service proxy using a known contract and then invoke operations on the proxy. The third task of the client is, for this discussion, the interesting task.

Using the reference returned from `get_account()`, the client will use the `Bus.createClient()` method to create a service proxy for the Account. The version of `Bus.createClient()` used to create a service proxy from a

reference takes two parameters: an Artix reference and the interface class for the referenced service. Example 117 shows the code for creating an Account service proxy from a reference.

Example 117: *Creating an Account Service Proxy*

```
acctProxy = bus.createClient(acctRef, Account);
```

Code for the client main()

Example 118 shows the completed code for the bank client's main line.

Example 118: *Code for Bank Client*

```
//Java
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;
import com.iona.schemas.references.Reference;

public class BankClient
{

  public static void main (String args[]) throws Exception
  {
1   Bus bus = Bus.init(args);

2   QName name = new QName("http://www.iona.com/bus/demos/bank",
                           "BankService");
3   String portName = new String("BankPort");

4   String wsdlPath = "file:/./bank.wsdl";
    URL wsdlURL = new File(wsdlPath).toURL();

5   Bank bankProxy = bus.createClient(wsdlURL, name, portName,
                                      Bank.class);

6   String account_name;
    System.out.println("What is the name of the account?");
    System.in.read(account_name);
```

Example 118:*Code for Bank Client*

```
7       Reference acctRef = bankProxy.get_account(account_name);

8       Account acctProxy = bus.createClient(acctRef, Account.class);

      // Invoke operations on acctProxy
        }
}
```

The code in Example 118 does the following:

1. Initializes the bus.
2. Creates the `QName` for the Bank service.
3. Sets the port name for the Bank service.
4. Sets the URL to the client's copy of the Bank service contract.
5. Creates a service proxy for the Bank service using `bus.createClient()`.
6. Gets the name of the account.
7. Gets a reference to the desired account by invoking `get_account()` on the Bank service proxy.
8. Uses the returned reference to create an Account service proxy using `bus.createClient()`.

Using References to Implement Callbacks

Overview

Another common use for Artix references is to create callbacks from a server to a client. When creating a callback, the client instantiates a servant object and registers it, using an Artix reference, with the server. The server can then create a service proxy for the client's callback object and invoke its operations to update the client.

For example, an accounts receivable system may need to notify its clients that it is closing the daily books and is not accepting new transactions until the operation is complete. In this case, the clients would have a callback object with two operations, `posting` and `done_posting`. The server would invoke posting to notify the client that it is not accepting new transactions. When it was done closing the books, the server would then invoke `done_posting`.

In this section

This section discusses the following topics:

The Accounting Contract	page 169
The Accounting Client	page 175
The Accounting Server	page 180

The Accounting Contract

Overview

The contract for an application the uses a callback needs to include the interface definition, binding definition, and service information for both the service implemented by the server and the callback object implemented by the client. When using callbacks the client essentially plays a dual role. It implements a servant, like a server process, and makes requests on a service.

Importing the reference schema

Any Artix service that uses references needs to include the XMLSchema definition for an Artix reference in its contract. This can be done in one of two ways. The most common way is to use an `<import>` element to import the XMLSchema definition that is provided with Artix. Example 110 shows a WSDL fragment that imports the reference schema.

Example 119: *Importing the Reference Schema*

```
<import namespace="http://schemas.iona.com/references"
        location="/usr/local/artix/schema/references.xsd" />
```

The other way is to add the reference definition directly into the contract.

You will also need to add an alias for the references namespace to the definitions tag at the top of the contract as shown in Example 111.

Example 120: *Reference Alias*

```
xmlns:reference="http://schemas.iona.com/references"
```

Messages with references

The `Register` interface's `register_callback` operation sends a reference to a `Notify` object. The message definition for the parameter of the operation has one part, `ref`, that is of type `reference:Reference`. Example 112 shows the definition for a message that contains a reference.

Example 121: *Message with a Reference*

```
<message name="regMessage">
  <part name="ref" type="references:Reference" />
</message>
```

The callback's interface

The interface for the callback object can be as complex or simple as your application requires. For this example, the callback object will only need two operations. One to inform the client that the server is busy and one to tell the client that the server is ready to receive new posts. Neither operation needs input or output messages, but because WSDL requires at least one `<input>` or `<output>` element the interface definition includes a dummy input message.

Example 122 shows the `<portType>` defining the callback object's interface.

Example 122: *Callback Interface*

```
<message name="callbackRequest" />
<portType name="Notify">
  <operation name="posting">
    <input name="param" message="tns:callbackRequest" />
  </operation>
  <operation name="done_posting">
    <input name="param" message="tns:callbackRequest" />
  </operation>
</portType>
```

Server interface

The server's interface needs one operation, `register_callback`, to register the client's callback object and create a proxy for it. In addition to the operation for registering the callback, the server can have any number of operations defined for providing services to the clients. In this example, the server has three operations: `deposit`, `withdraw`, and `dailyPosting`. The client shown in this example only invokes `desposit` and `withdraw`. An administrative client invokes `dailyPosting`.

Example 123 shows the <portType> defining the server's interface.

Example 123: *Server Interface*

```
<portType name="Register">
  <operation name="register_callback">
    <input name="param" message="tns:refMessage" />
  </operation>
  <operation name="deposit">
    <input name="amount" message="tns:amtMessage" />
    <output name="return" message="tns:amtMessage" />
  </operation>
  <operation name="withdraw">
    <input name="amount" message="tns:amtMessage" />
    <output name="return" message="tns:amtMessage" />
  </operation>
  <operation name="dailyPosting">
    <input name="date" message="tns:dateMessage" />
  </operation>
</portType>
```

Bindings

The callback object's interface can be bound to any of the message formats supported by Artix. Because the server's interface includes an operation that has a reference as a parameter, it can only be bound to a SOAP message or a CORBA message. In this example, both interfaces are bound to SOAP messages.

Transport details

Because both the callback object and the server are registered as static servants, they can use any of the transports supported by Artix. In this example, HTTP is used.

Contract

Example 124 shows the complete contract used for the code generated in the following discussions about callbacks.

Example 124:*Callback Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/bus/demos/callbacks"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.iona.com/transports/http"
  xmlns:references="http://schemas.iona.com/references"
  targetNamespace="http://www.iona.com/bus/demos/callbacks"
  name="BankService">
  <import namespace="http://schemas.iona.com/references"
          location="/usr/local/artix/schema/references.xsd" />
  <message name="amtMessage">
    <part name="amount" type="xsd:float"/>
  </message>
  <message name="amtResponse">
    <part name="return" type="xsd:float"/>
  </message>
  <message name="refMessage">
    <part name="ref" type="references:Reference"/>
  </message>
  <message name="dateMessage">
    <part name="date" type="xsd:string"/>
  </message>
<message name="callbackRequest" />
  <portType name="Notify">
    <operation name="posting">
      <input name="param" message="tns:callbackRequest" />
    </operation>
    <operation name="done_posting">
      <input name="param" message="tns:callbackRequest" />
    </operation>
  </portType>
```

Example 124:*Callback Contract*

```
<portType name="Register">
  <operation name="register_callback">
    <input name="param" message="tns:refMessage" />
  </operation>
  <operation name="deposit">
    <input name="amount" message="tns:amtMessage" />
    <output name="return" message="tns:amtResponse" />
  </operation>
  <operation name="withdraw">
    <input name="amount" message="tns:amtMessage" />
    <output name="return" message="tns:amtResponse" />
  </operation>
  <operation name="dailyPosting">
    <input name="date" message="tns:dateMessage" />
  </operation>
</portType>
<binding name="NotifyBinding" type="tns:Notify">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="posting">
    <soap:operation soapAction="http://www.iona.com/bus/demos/callbacks" style="rpc"/>
    <input>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/callbacks"/>
    </input>
  </operation>
  <operation name="done_posting">
    <soap:operation soapAction="http://www.iona.com/bus/demos/callbaks" style="rpc"/>
    <input>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/callbacks"/>
    </input>
  </operation>
</binding>
<binding name="RegisterBinding" type="tns:Register">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="register_callback">
    <soap:operation soapAction="http://www.iona.com/bus/demos/callbacks" style="rpc"/>
    <input>
      <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="http://www.iona.com/bus/demos/callbacks"/>
    </input>
  </operation>
```

Example 124:*Callback Contract*

```
    <operation name="deposit">
      <soap:operation soapAction="http://www.iona.com/bus/demos/callbacks" style="rpc"/>
      <input>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/callbacks"/>
      </input>
      <output>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/callbacks"/>
      </output>
    </operation>
    <operation name="withdraw">
      <soap:operation soapAction="http://www.iona.com/bus/demos/callbacks" style="rpc"/>
      <input>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/callbacks"/>
      </input>
      <output>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/callbacks"/>
      </output>
    </operation>
    <operation name="dailyPosting">
      <soap:operation soapAction="http://www.iona.com/bus/demos/callbacks" style="rpc"/>
      <input>
        <soap:body use="literal" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://www.iona.com/bus/demos/callbacks"/>
      </input>
    </operation>
  </binding>
  <service name="NotifyService">
    <port name="NotifyPort" binding="tns:NotifyBinding">
      <soap:address location="http://localhost:0"/>
    </port>
  </service>
  <service name="RegisterService">
    <port name="RegisterPort" binding="tns:RegisterBinding">
      <soap:address location="http://localhost:0/RegisterService/RegisterPort/"/>
    </port>
  </service>
</definitions>
```

The Accounting Client

Overview

A client that has a callback object has two major parts to develop:

- The callback object's implementation object.
- The client's `main()` that performs the clients work.

When using a callback, the client's `main()` will perform one additional task that is normally only performed in servers. It will instantiate a servant for the callback object and register it with the bus.

Callback implementation

The callback object for this example is very simple. It has one static member, `busy`, that is set to `1` when `posting()` is invoked and set to `0` when `done_posting()` is invoked. Using the instance of `NotifyImpl` registered with the bus in the client's `main()`, you can check the value of `busy` to see if the `Register` service is doing its daily posting and not accepting new requests.

To avoid thread conflicts, the callback object's methods are synchronized. When the methods complete, they then notify all interested parties that callback object has been modified. This notifies the client the status has been updated and it can stop waiting for the server.

Example 125 shows the code for the callback object.

Example 125: *Callback Object*

```
package com.iona.bus.demos.callbacks;

import java.net.*;
import java.rmi.*;

public class NotifyImpl implements java.rmi.Remote
{
  public int busy = 0;
```

Example 125:*Callback Object*

```
  public void posting()
  {
    synchronize(this)
    {
      busy = 1;
      notifyAll();
     }
  }

  public void done_posting()
  {
    synchronize(this)
    {
      busy = 0;
      notifyAll();
     }
  }
}
```

The client main()

The client `main()` in this example does six things:

1. Creates a service proxy for the `Register` service.
2. Creates a servant for the callback object.
3. Registers the callback object's servant with the bus so that it can receive requests.
4. Registers the callback object with the `Register` service.
5. Invokes operations on the `Register` service.
6. Checks the callback object to see if the `Register` service is posting.

Example 126 shows the code for client `main()`.

Example 126: *Callback Client Main()*

```
//Java
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;
import com.iona.schemas.references.Reference;

public class RegisterClient
{

  public static void main (String args[]) throws Exception
  {
    char op;
1   Bus bus = Bus.init(args);

2   QName name = new
          QName("http://www.iona.com/bus/demos/callbacks",
                "RegisterService");
    String portName = new String("RegisterPort");

    String wsdlPath = "file:/./resister.wsdl";
    URL wsdlURL = new File(wsdlPath).toURL();

    Register registerProxy = bus.createClient(wsdlURL, name,
                                              portName,
                                              Register.class);

3   NotifyImpl notify = new NotifyImpl();

    String contract = new String("./register.wsdl");

4   Servant servant = new SingleInstanceServant(notify, contract,
                                                bus);

    QName notifyName = new
               QName("http://www.iona.com/bus/demos/callbacks",
                     "NotifyService");
```

Example 126:*Callback Client Main()*

```
5     bus.registerServant(servant, notifyName);

6     Reference ref = bus.createReference(notifyName);

7     registerProxy.register_callback(ref);

    Float amount;
    float balance;
    String temp;

     while(true)
     {

8      synchronize(notify)
       {
9        while(notify.busy == 1)
         {
           System.out.println("The Server is posting. Please
                               wait.");
10         notify.wait();
         }
       }

11     System.out.println("Choose an option:");
       System.out.println("1) Deposit");
       System.out.println("2) Withdraw");
       System.out.println("3) Exit");
       System.in.read(op);

       switch(op)
       {
         case '1':
           System.out.println("Amount to deposit?");
           System.in.read(temp);
           amount = new Float(temp);
           balance = registerProxy.deposit(amount.floatValue());
           System.out.println("New balance: "+balance);
           break;
```

Example 126:*Callback Client Main()*

```
        case '2':
          System.out.println("Amount to withdraw?");
          System.in.read(temp);
          amount = new Float(temp);
          balance = registerProxy.withdraw(amount.floatValue());
          System.out.println("New balance: "+balance);
          break;
        Case '3':
          return;
       }
    }
  }
}
```

The code in Example 126 does the following:

1. Initializes a bus for the client.
2. Creates a proxy for the `Register` service.
3. Creates an instance of `NotifyImpl` to be the callback object.
4. Creates a servant to wrap the callback object.
5. Registers the servant with the bus.
6. Creates a reference for the callback object's servant.
7. Registers the callback by invoking the `Register` service's `register_callback()` operation.
8. Ensures that the callback object cannot be modified by other threads before checking its state.
9. If the callback object's busy flag is set to 1 the server is doing its daily posting and the client needs to wait.
10. Waits on the callback object. When the server changes the value of busy, this call will stop blocking and the flag can be checked again.
11. Makes requests on the `Register` service.

The Accounting Server

Overview

The server in this example also exhibits some hybrid behavior. The `register_callback` operation receives a reference to the client's callback object and creates a service proxy for it. In this example, the proxy is put into an object-level data element and the `dailyPosting` operation invokes the proxy's operations to inform the clients when the server is posting.

Server main()

In this example, the server's `main()` is a standard Artix server `main()`. It initializes a bus instance, registers a `Servant` that wraps an instance of `RegisterImpl`, and then calls `Bus.run()`. For a discussion of writing an Artix server `main()` see "Developing a Server" on page 15.

RegisterImpl

The Accounting server's implementation object, as generated by `wsdltojava`, is called `RegisterImpl`. It has four methods: `register_callback()`, `dailyPosting()`, `deposit()`, and `withdraw()`. `deposit()` and `withdraw()` perform data requests for the client and they are left for you to implement.

For the discussion of callbacks, `register_callback()` and `dailyPosting()` are of interest. `register_callback()` is responsible for receiving the callback object's reference and instantiating a proxy for it. In this example, the proxy is stored in the objects `notify` member. `dailyPosting()` then invokes the callback object's operations to inform the client when the server is busy.

Example 127 shows the completed `RegisterImpl` class. The code in bold is added to the generated class by the user.

Example 127: *RegisterImpl*

```
package com.iona.bus.demos.callbacks;

import java.net.*;
import java.rmi.*;

import com.iona.schemas.references.Reference;
import com.iona.jbus.*;
import java.lang.String;
```

Example 127: *RegisterImpl*

```
public class RegisterImpl implements java.rmi.Remote
{
  NotifyImpl notify;

  public void register_callback(com.iona.schemas.references.Reference ref)
  {
    com.iona.jbus.Bus bus = DispatchLocals.getCurrentBus();

    notify = bus.createClient(ref, Notify.class);
  }

    public float deposit(float ammount)
    {
        // User code goes in here.
        return 0.0f;
    }

    public float withdraw(float ammount) {
        // User code goes in here.
        return 0.0f;
    }

  public void dailyPosting(String date)
  {
    notify.posting();

    // User code goes in here.

   notify.done_posting();
  }
}
```

register_callback()

`register_callback()` does the following:

1. Gets a handle on the bus hosting this servant.
2. Creates a proxy for the callback object using the reference sent by the client.

dailyPosting()

`dailyPosting()` does the following:

1. Invokes the callback object's `posting` operation to notify the client that the server is busy.
2. Performs the tasks involved in closing the daily books and posting the results. This logic is left to the user to implement.
3. When the daily posting tasks are complete, it invokes the callback object's `done_posting` operation to notify the client that the server is ready to handle new requests.

CHAPTER 9

The Artix Locator

The Artix locator is a central repository for storing references to Artix endpoints. If you set up your Artix servers to register their endpoints with the locator, you can code your clients to open server connections by retrieving endpoint references from the locator.

In this chapter

This chapter discusses the following topics:

Overview of the Locator	page 184
Locator WSDL	page 187
Registering Endpoints with the Locator	page 191
Reading a Reference from the Locator	page 192

Overview of the Locator

Overview

The Artix locator is a service which can optionally be deployed for the following purposes:

- *Repository of endpoint references*—endpoint references stored in the locator enable clients to establish connections to Artix services.
- *Load balancing*—if multiple service instances are registered against a single service name, the locator load balances over the different service instances using a round-robin algorithm.

Figure 5 gives a general overview of the locator architecture.

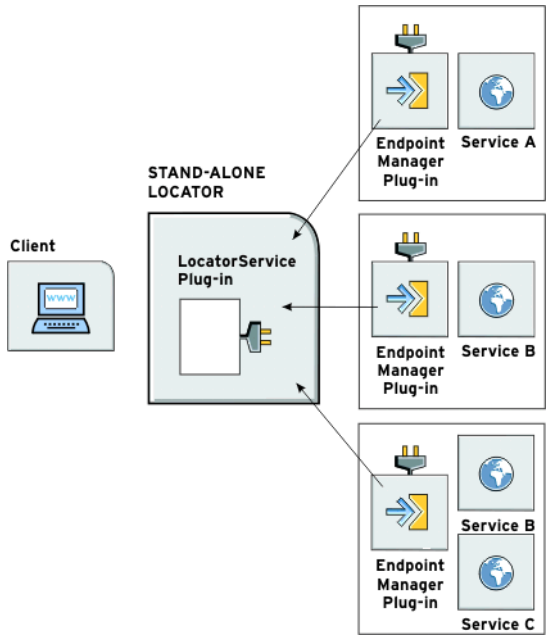

Figure 5: *Artix Locator Overview*

Locator service

There are two basic options for deploying the locator service, as follows:

- *Standalone deployment*—the locator is deployed as an independent server process (as shown in Figure 5). This approach is described in detail in *Deploying and Managing Artix Solutions*. Sample source code for such a standalone locator service is provided in the `demos/uncategorized/locator` demonstration.
- *Embedded deployment*—the locator is deployed by embedding it within another Artix server process. This approach is possible because the locator is implemented as a plug-in, which can be loaded into any Artix application.

Endpoint definition

An Artix *endpoint* is a particular WSDL service (identified by a service name) in a particular bus instance (identified by a WSDL location URL). Hence, it is possible to have endpoints with the same service type and service name, as long as they are registered with different bus instances. A WSDL location URL and a service name together identify an endpoint.

Registering endpoints

A server that loads the locator's endpoint manager plugin automatically registers its endpoints with the locator in order to make them accessible to Artix clients. When a server registers an endpoint in the locator, it creates an entry in the locator that associates a service name with an Artix reference for that endpoint.

Looking up references

An Artix client looks up a reference in the locator in order to find an endpoint associated with a particular service. After retrieving the reference from the locator, the client can then establish a remote connection to the relevant server by instantiating a client proxy object. This procedure is independent of the type of binding or transport protocol.

Load balancing with the locator

If multiple endpoints are registered against a single service name in the locator, the locator will employ a round-robin algorithm to pick one of the endpoints. Hence, the locator effectively *load balances* a service over all of its associated endpoints.

For example, Figure 5 on page 184 shows `Service A` with two endpoints registered against it. When the Artix client looks up a reference for `Service A`, it obtains a reference to whichever endpoint is next in the sequence.

Locator WSDL

Overview

The locator WSDL contract, `locator.wsdl`, defines the public interface of the locator through which the service can be accessed. This section shows extracts from the locator WSDL that are relevant to normal user applications. The following aspects of the locator WSDL are described here:

- Binding and protocol
- WSDL contract
- Java mapping

Binding and protocol

The locator service is accessed using the SOAP binding and the HTTP transport.

WSDL contract

Example 128 shows an extract from the locator WSDL contract that focuses on the aspects of the contract relevant to an Artix application programmer. There is just one WSDL operation, `lookup_endpoint`, that an Artix client typically needs to call.

Example 128: *Locator WSDL Contract*

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:ref="http://schemas.iona.com/references"
   xmlns:ls="http://ws.iona.com/locator"
   targetNamespace="http://ws.iona.com/locator">
  <types>
    <xs:schema targetNamespace="http://ws.iona.com/locator">
1     <xs:import schemaLocation="../../../schemas/references.xsd"
                 namespace="http://schemas.iona.com/references"/>
      ...
2     <xs:element name="lookupEndpoint">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="service_qname" type="xs:QName"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
```

Example 128: *Locator WSDL Contract*

```
3       <xs:element name="lookupEndpointResponse">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="service_endpoint"
                          type="ref:Reference"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:complexType name="EndpointNotExistFaultException">
          <xs:sequence>
            <xs:element name="error" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
4       <xs:element name="EndpointNotExistFault"
                    type="ls:EndpointNotExistFaultException"/>
    </xs:schema>
  </types>
    ...
  <message name="lookupEndpointInput">
    <part name="parameters" element="ls:lookupEndpoint"/>
  </message>
  <message name="lookupEndpointOutput">
    <part name="parameters"
          element="ls:lookupEndpointResponse"/>
  </message>
  <message name="endpointNotExistFault">
    <part name="parameters" element="ls:EndpointNotExistFault"/>
  </message>
5 <portType name="LocatorService">
6   <operation name="lookup_endpoint">
      <input message="ls:lookupEndpointInput"/>
      <output message="ls:lookupEndpointOutput"/>
      <fault name="fault" message="ls:endpointNotExistFault"/>
    </operation>
  </portType>
...
  <service name="LocatorService">
    <port name="LocatorServicePort"
          binding="ls:LocatorServiceBinding">
7     <soap:address  location="http://localhost:0"/>
    </port>
  </service>
</definitions>
```

The preceding locator WSDL extract can be explained as follows:

1. This line imports the schema definition of the `ref:Reference` type. You might have to edit the value of the `schemaLocation` attribute, if the `references.xsd` schema file is stored in a different location relative to the `locator.wsdl` file.
2. The `lookupEndpoint` type is the input parameter type for the `lookup_endpoint` operation. It contains just the QName (qualified name) of a particular WSDL service.
3. The `lookupEndpointResponse` type is the output parameter type for the `lookup_endpoint` operation. It contains an Artix reference for the specified service. If more than one endpoint is registered against a particular service name, the locator picks one of the endpoints using a round-robin algorithm.
4. The `EndpointNotExist` fault would be thrown if the `lookup_endpoint` operation fails to find an endpoint registered against the requested service type.
5. The `LocatorService` port type defines the public interface of the Artix locator service.
6. The `lookup_endpoint` operation, which is called by Artix clients to retrieve endpoint references, is the only operation from the `LocatorService` port type that user applications would typically need.
7. The SOAP `location` attribute specifies the host and IP port for the locator service. If you want the locator to run on a different host and listen on a different IP port, you should edit this setting.

Java mapping

Example 129 shows an extract from the Java mapping of the `LocatorService` port type. This extract shows only the `lookup_endpoint` operation—the other operations defined for the locator are normally not needed by user applications.

Example 129: *Java Mapping of the LocatorService Port Type*

```
// Java
package com.iona.ws.locator;
```

Example 129:*Java Mapping of the LocatorService Port Type*

```
import java.net.*;
import java.rmi.*;

import com.iona.schemas.references.Reference;
import javax.xml.rpc.holders.StringHolder;
import java.lang.String;
import javax.xml.namespace.QName;

/**
 * com.iona.ws.locator.LocatorService
 */
public interface LocatorService extends java.rmi.Remote
{
...

  public com.iona.schemas.references.Reference
    lookup_endpoint(javax.xml.namespace.QName service_qname)
  throws EndpointNotExistFaultException, RemoteException;
}
```

Registering Endpoints with the Locator

Overview

To register a server's endpoints with the locator, you must configure the server to load a specific set of plug-ins. Once the appropriate plug-ins are loaded, the server will automatically register every endpoint that it creates.

Configuring a server to register endpoints

A server that is to register its endpoints with the locator must be configured to include the `soap`, `http`, and `locator_endpoint` plug-ins, as shown in Example 130.

Example 130: *Server Configuration Scope for Using the Locator*

```
# Artix Configuration File (artix.cfg)
...
located_server
{
  orb_plugins = ["xmlfile_log_stream", "soap", "http",
                 "locator_endpoint"];
  plugins:locator:wsdl_url="../wsdl/locator.wsdl";
};
```

When running the server, remember to select the appropriate configuration scope by passing it as the `-ORBname` command-line parameter.

References

For more details about configuring a server to register endpoints, see the following references:

- The chapter on using the locator in *Deploying and Managing Artix Solutions*.
- The `locator` demonstration in `artix/Version/demos/advanced/locator`.

Reading a Reference from the Locator

Overview

After the target server has started up and registered its endpoints with the locator, an Artix client can then lookup the server's endpoints using the locator. The client can then connect to the target server by creating a service proxy using the reference from the locator. Figure 6 shows an outline of how a client connects to a server in this way.

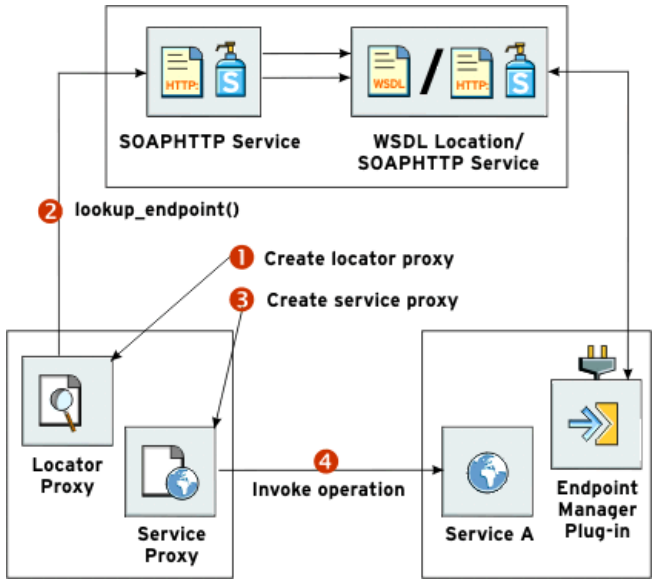

Figure 6: *Steps to Read a Reference from the Locator*

Programming steps

The main programming steps needed to read a reference from the locator, as shown in Figure 6, are as follows:

1. Generate the types and the interface for the locator by running `locator.wsdl` through `wsdltojava`.

   Note: This only needs to be done the first time you want to build a client to use the locator. The generated Java code can be built into a class and reused for subsequent client applications.

2. Construct a locator service proxy.
3. Use the locator proxy to invoke `lookup_reference()`.
4. Use the reference returned from `lookup_reference()` to construct a proxy to the service.
5. Invoke an operation using the created service proxy.

Example

Example 131 shows an example of the code for an Artix client that retrieves a reference to a `SimpleService` service from the Artix locator.

Example 131: *Example of Reading a Reference from the Locator Service*

```
//Java
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;
import com.iona.schemas.references.Reference;
import com.iona.ws.locator.*;

public class SimpleServiceClient
{

  public static void main (String args[]) throws Exception
  {

1   Bus bus = Bus.init(args);
```

Example 131: *Example of Reading a Reference from the Locator Service*

```
2     QName name = new QName("http://ws.iona.com/locator",
                             "LocatorService");

3    QName lookup_name = new QName("http://www.iona.com/bus/tests",
                                   "SOAPHTTPService");
4    QName portName = new QName("","LocatorServicePort");

5   // Build the Locator Service Proxy
      String wsdlPath = "file:/../wsdl/locator.wsdl";
      URL wsdlLocation = new File(wsdlPath).toURL();

      ServiceFactory factory = ServiceFactory.newInstance();
      Service service = factory.createService(wsdlLocation, name);

      LocatorService locator =
        (LocatorService)service.getPort(portName,
                                        LocatorService.class);

6     //Invoke lookup_endpoint()
      Reference simp_ref = locator.lookup_endpoint(lookup_name);

7     // Build a proxy to the target service from the reference
      SimpleService simple_client =
       (SimpleService)bus.createClient(simp_ref,
                                       SimpleService.class);

8     String greeting = "Greetings from a located client";
      String result;
      result = simple_client.say_hello(greeting);
      System.out.println("say_hello method returned: "+result);
    }
}
```

The code in Example 131 can be explained as follows:

1. You should ensure that the client picks up the correct configuration by passing the appropriate value of the `-ORBname` parameter.
2. This line constructs a QName, `name`, that identifies the `<service name="LocatorService">` tag from the locator WSDL. See the listing of the locator WSDL in Example 128 on page 187.
3. This line constructs a QName, `lookup_name`, that identifies the `SOAPHTTPService` service from the `SimpleService` WSDL.

4. This port name refers to the `<port name="LocatorServicePort" ...>` tag in the locator WSDL (see Example 128 on page 187).
5. The locator service proxy is created by using the standard JAX-RPC method for creating a dynamic proxy. For details see "Developing a Client" on page 20.
6. The `lookup_endpoint()` operation is invoked on the locator to find an endpoint of `SOAPHTTPService` type.

   Note: If there is more than one WSDL port registered for the `SOAPHTTPService` server, the locator service employs a round-robin algorithm to choose one of the ports to use as the returned endpoint.

7. The `SimpleService` reference returned from the locator, `simp_ref`, is then passed to the bus' `createClient()` proxy constructor. The `createClient()` proxy constructor takes `Reference` type and the class of the proxy to be created as its arguments.
8. You can now use the simple client proxy to make invocations on the remote Artix server.

CHAPTER 10

Using Message Contexts

Artix implements and extends the J2EE MessageContext interface to allow users to manipulate metadata about messages and transports.

In this chapter

This chapter discusses the following topics:

Understanding Message Contexts in Artix	page 198
Sending Header Information Using Contexts	page 218

Understanding Message Contexts in Artix

Overview

Artix implements the JAX-RPC `MessageContext` interface. JAX-RPC message contexts are primarily used in writing message handlers, but can also be used to store metadata about messages or pass state information into or out of the message handling chain. Generally, this metadata is not passed across the wire with the message.

In addition, Artix extends the JAX-RPC message contexts to provide a consistent, thread safe mechanism for passing additional information along with request and reply messages. Currently, this mechanism can be used to send SOAP headers and security information when using the SOAP binding.

Contexts and the bus

Message contexts are bus objects that application level code can access. To manage the Artix message contexts associated with it, a bus uses a context registry that allows it to instantiate thread specific message contexts. Using the message context, application code can access any of the properties set by the application. Because the contexts are thread specific bus objects, any changes made to a property stored in a context by a message handler is reflected at the application level.

Artix message contexts, because they hold information which is to be written out on the wire, have a request context container and a reply context container for the thread in which it is running. The reply context container

stores information returned from a server and the request context container stores information that is sent along with a request. This is shown in Figure 7.

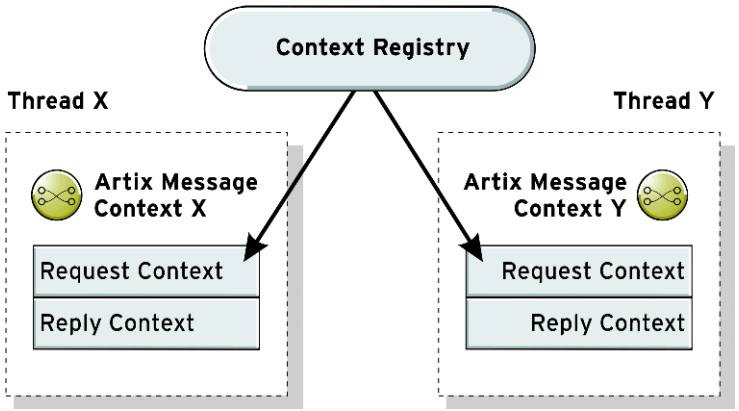

Figure 7: *Overview of the Message Context Mechanism*

Getting message contexts

To access message contexts in your application do the following:

1. If you are using Artix message contexts, register the type factories for the data stored in the contexts. See "Registering Type Factories" on page 128.
2. Get a reference to the bus' context registry.
3. Get the message context for the thread in which your application is running from the context registry.

Working with message contexts

Once you have gotten the message context, you can chose to use it as a generic JAX-RPC message context or to cast it to an Artix message context. Both interfaces will allow you to access all of the properties set for the active bus, but the Artix message context simplifies the accessing Artix specific properties. The Artix message context interface is an extension of the generic message context interface, so all of the generic message context methods are available after you cast a generic message context to an Artix message context.

In this section

This section discusses the following topics:

Getting the Context Registry	page 201
Getting the Message Context for a Thread	page 203
Working with Generic Contexts	page 206
Working with Artix Message Contexts	page 211

Getting the Context Registry

Overview

The *Context Registry* is maintained by the bus. It contains an entry for all of the Artix specific property types registered with the bus. It also instantiates thread specific message contexts and hands out references to the proper message context to requesting applications.

Procedure

The `Bus` has a method, `getContextRegistry()`, that returns a reference to the bus instance's context registry. The context registry is an object of type `ContextRegistry`. Example 132 shows the signature of `getContextRegistry()`. Because the context registry is specific to an instantiated bus instance, you must call it on an initialized bus instance.

Example 132: *getContextRegistry()*

```
ContextRegistry com.iona.jbus.Bus.getContextRegistry();
```

Example

Example 133 shows an example of getting the context registry from within the implementation object of an Artix service.

Example 133: *Getting the Context Registry*

```
//Java
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
2 com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3 ContextRegistry contReg = def_bus.getContextRegistry();

...
}
```

The code in Example 133 does the following:

1. Import the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Get a handle to the application's bus.
3. Call `getContextRegistry()` on the default bus to get the default bus' context registry.

Getting the Message Context for a Thread

Overview

To ensure thread safety, the context registry creates a *Message Context* object for each thread. The message contexts maintained by the context registry are passed as generic J2EE `MessageContext` objects. These objects provide access to properties stored in the contexts using the APIs defined in the J2EE specification.

Artix provides two means of getting the currnet message context for a thread. If you have the context registry, you can use the registry's `getCurrent()` method. If you do not have the context registry, you can use the `DispatchLocals.getCurrentContext()` method.

To manipulate Artix specific properties you must cast the returned `MessageContext` into an `IonaMessageContext` object. Once the `MessageContext` is cast to an `IonaMessageContext` it is an Artix message context. The Artix message context APIs provide easy access to Artix specific properties and track the context container for which each property is set.

getCurrent()

Message contexts are passed out by the context registry using the registry's `getCurrent()` method. `getCurrent()` returns the message context object for the thread from which it is called. Message contexts are returned as generic J2EE `MessageContext` objects. Example 134 shows the signature for `getCurrent()`.

Example 134: *getCurrent()*

```
javax.xml.rpc.handler.MessageCcontext ContextRegistry.getCurrent();
```

If you want to use the returned message context to work with Artix specific context information you can cast it to an `IonaMessageContext` object. The `IonaMessageContext` object is discussed in "Working with Artix Message Contexts" on page 211.

Example 135 shows how to get an message context from the context registry.

Example 135:*Getting a Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
import javax.xml.rpc.*

1  import com.iona.jbus.*;

public class Atherny
{
2  com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3  ContextRegistry contReg = def_bus.getContextRegistry();

4  MessageContext messCont = contReg.getCurrent();
...
}
```

The code in Example 135 does the following:

1. Import the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Get a handle to the application's bus.
3. Call `getContextResistry()` on the default bus to get the default bus' context registry.
4. Call `getCurrent()` on the context registry to get the Artix message context for the application's thread.

DispatchLocals

`DispactLocals` is a globally accessable interface that provides a simple method for getting the current message context for a thread. Its `getCurrentMessageContext()` method returns the message context object for the thread from which it is called. Message contexts are returned as generic J2EE `MessageContext` objects. Example 136 shows the signature for `getCurrent()`.

Example 136:*getCurrentMessageContext()*

```
javax.xml.rpc.handler.MessageCcontext getCurrentMessageContext();
```

If you want to use the returned message context to work with Artix specific context information you can cast it to an `IonaMessageContext` object. The `IonaMessageContext` object is discussed in "Working with Artix Message Contexts" on page 211.

Example 137 shows how to get an message context using the `DispatchLocals` interface.

Example 137: *Getting a Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
import javax.xml.rpc.*

import com.iona.jbus.*;

public class Atherny
{
  MessageContext messCont = DispatchLocals.getCurrentBus();
  ...
}
```

Working with Generic Contexts

Overview

A JAX-RPC message context is a container for properties that are shared among the participants in applications message handling chain. They have some predefined properties that are made available to message handlers that run below the application level. However, you can add any named property you like to the context as long as the name does not conflict with one of the predefined properties.

Properties set in the message context are only available at certain steps along the message handling chain. Properties set in the context by message handlers are only available to message handlers further down the processing chain and are destroyed once the operation's invocation completes. Properties set at the application level are available globally and live for the duration of the application.

Generic message contexts have methods to set a property in the context, to get a property from the context, and to remove a property from the context. They also have methods to determine what properties are set in the context.

Setting a property in the context

Before a property exists in the message context it must be set using the message context's `setProperty()` method. Example 138 shows the signature for `setProperty()`. The first parameter, `name`, can be any string as long as it is unique among the properties set in the context. The second parameter, `value`, can be any instantiated Java object. It becomes the value of the property stored in the context.

Example 138: *MessageContext.setProperty()*

```
void setProperty(Sting name, Object value);
```

The scope of the property depends on where in the message handling chain the property is set into the context. Properties set at the level from which the operations are invoked they are global in scope and exist for the duration of the process' lifecycle or until they are explicitly removed from the message context. Properties set by message handlers are only available to message handlers further down the message handler chain and expire once the operation's invocation is completed. For more information about message handlers, see "Writing Message Handlers" on page 239.

Example 139 shows the code for setting a property in the request context.

Example 139: *Setting a Property in a Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
2 com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3 ContextRegistry contReg = def_bus.getContextRegistry();

4 MessageContext context = contReg.getCurrent();

5 boolean isEncrytped = TRUE;

6 context.setProperty("isEncrypted", isEncrypted);

...
}
```

The code in Example 139 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Gets a handle to the application's bus.
3. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
4. Calls `getCurrent()` on the context registry to get the message context for the application's thread.
5. Creates the an instance of the property's class and set the values.
6. Sets the property by calling `setProperty()`.

Getting a property from the context

You get a property's value from the message context using its `getProperty()` method. Example 140 shows the signature for `getProperty()`. It takes a single parameter, `name`, that is the name of the property for which you want the value. If the property exists, it is returned. If the property does not exist, `null` is returned.

Example 140: *MessageContext.getProperty()*

```
Object getProperty(String name);
```

Example 141 shows the code for getting a SOAP header from the request context.

Example 141: *Getting a Property from the Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
2 com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3 ContextRegistry contReg = def_bus.getContextRegistry();

4 MessageContext context = contReg.getCurrent();

5 boolean encrypt = (boolean)context.getProperty("isEncrypted");
...
}
```

The code in Example 141 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Gets a handle to the application's bus.
3. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
4. Calls `getCurrent()` on the context registry to get the message context for the application's thread.
5. Gets the property by calling `getProperty()` with the appropriate name.

Removing a property from the context

If you wish to remove a property from the message context, you do so using the message context's `removeProperty()` method. Example 142 shows the signature for `removeProperty()`. It takes a single parameter, `name`, that represents the name of the property you wish to remove.

Example 142: *MessageContext.removeProperty()*

```
void removeProperty(String name);
```

Example 143 shows the code for removing a property from the message context.

Example 143: *Removing a Property from a Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
import com.iona.jbus.*;

public class Atherny
{
1 com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

2 ContextRegistry contReg = def_bus.getContextRegistry();

3 MessageContext context = contReg.getCurrent();

4 context.removeProperty("isEnctryted");
...
}
```

The code in Example 143 does the following:

1. Gets a handle to the application's bus.
2. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
3. Calls `getCurrent()` on the context registry to get the message context for the application's thread.
4. Removes the property by calling `removeProperty()`.

Determining what properties are set

The JAX-RPC `MessageContext` interface has two methods that allow you to determine what properties are set in a context. `containsProperty()` takes the name of a property, as a `String`, and returns true if the property is set and false if the property is not. `getPropertyNames()` returns an `Iterator` object with the names of all properties stored in the message context.

Example 144 shows the code for seeing if a property is set in the message context.

Example 144: *Querying a Property in the Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
import com.iona.jbus.*;

public class Atherny
{
com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

ContextRegistry contReg = def_bus.getContextRegistry();

MessageContext context = contReg.getCurrent();

if (context.containsProperty("isEnctryted"))
 {
    System.out("The property is set");
 }
...
}
```

Working with Artix Message Contexts

Overview

Each Artix message context holds one *Request Context Container* and one *Reply Context Container*. The request context container holds all of the properties associated with messages that originate as service requests in a proxy. The reply context container holds all of the properties associated with messages that are created by services in response to a request. In both instances, the properties in the context container are passed all the way through the request and reply chain. For example, if `Client` makes a request on `ServerA`, `ServerA` would receive the properties set in the request context from the client. If `ServerA` then passes the request along to `ServerB`, `ServerB` also receives the request context sent by `Client`. The same is true when using the Artix router. Figure 8 shows how context properties are passed with messages.

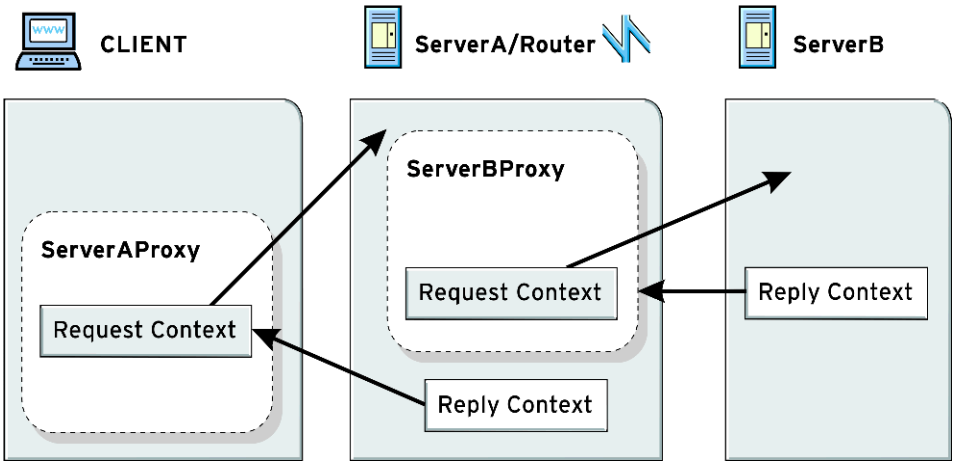

Figure 8: *Contexts Passed Along Request/Reply Chain*

The context containers hold the data for all of the contexts instantiated in the Artix message context's thread. Each context container can hold one instance of a registered property type. Properties are instantiated separately for the request context container and the reply context container. For instance, you can get a SOAP header property for the request context

container and leave the reply context container empty. In that case, the SOAP header property would be included in all request messages sent from the thread in which it was set.

Setting a property

Before you can get a property from one of the context containers, the property must be set in that container. Properties are set in one of two ways. The first is that the property is set by the sender of the message. For example, if a client sends a request with a WS-Security header, the server's request context container will have the WS-Security property set.

The second is to use the message context's setter methods. The message context has four setter methods: `setReplyContext()`, `setReplyContextAsString()`, `setRequestContext()`, and `setRequestContextAsString()`. `setReplyContext()` and `setRequestContext()` allow you to set the values for properties that are defined as non-string data. `setReplyContextAsString()` and `setRequestContextAsString()` allow you to set the values for properties that are defined as strings. Example 145 shows the signature for these methods.

Example 145: *Methods for Setting a Property*

```
void setReplyContext(QName name, Object value);
void setReplyContextAsString(QName name, String value);
void  setRequestContext(QName name, Object value);
void setRequestContextAsString(QName name, String value);
```

The first parameter to these methods, `name`, specifies the name of the property you desire to set. The `QName` passed in must be a `QName` of a property that is registered with the context registry.

The second parameter, `value`, is data you are using to set the property. It must be of the appropriate type for the property specified in `name`.

To set a property do the following:

1. Create an instance of the object representing the property you want to set.
2. Set the desired fields of the newly created property.

3. Call the appropriate setter method with the name of the property you are setting and the property instance you created. For example, to set a property into the reply context container, you would use `setReplyContext()`.

Example 146 shows the code for setting a property in the request context.

Example 146: *Setting a Property in an Artix Message Context*

```
//Java
import java.net.*;
import java.rmi.*;
1  import com.iona.jbus.*;

public class Atherny
{
2  com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3  ContextRegistry contReg = def_bus.getContextRegistry();

4  IonaMessageContext context =
     (IonaMessageContext)contReg.getCurrent();

5  clientType HTTPClientAttrs = new clientType();
   HTTPClientAttrs.set_username("Murphy");
   HTTPClientAttrs.set_password("1234");

6  QName contextName = new QName("http://widgets.com/",
                                 "HTTPClientAttributes");

7  context.setRequestContext(contextName, HTTPClientAttrs);

...
}
```

The code in Example 146 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Gets a handle to the application's bus.
3. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
4. Calls `getCurrent()` on the context registry to get the message context for the application's thread and casts it to an Artix message context.

5. Creates the an instance of the property's class and set the values.
6. Creates the QName for the property.
7. Sets the property by calling `setRequestContext()` with the appropriate `QName` and the newly created property object.

Getting a property

Artix message contexts have four methods that allows you to get a property from one of the context containers. These methods are `getReplyContext()`, `getReplyContextAsString()`, `getRequestContext()`, and `getRequestContextAsString()`. `getReplyContext()` and `getRequestContext()` return the property a generic Java `Object` that must be cast into the property's type. `getReplyContextAsString()` and `getRequestContextAsString()` return the values of properties of type `String`. Example 147 shows the signature for these methods.

Example 147: *Methods for Getting a Property*

```
Object getReplyContext(QName name);
String getReplyContextAsString(QName name);
Object getRequestContext(QName name);
String getRequestContextAsString(QName name);
```

They take a single parameter, `name`, that specifies the name of the property you desire to get. The `QName` passed in must be a `QName` of a property that is registered with the context registry. You can register your own properties to use as SOAP headers.

Example 148 shows the code for getting a SOAP header from the request context.

Example 148: *Getting a Property*

```
//Java
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
2 com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3 ContextRegistry contReg = def_bus.getContextRegistry();
```

Example 148: *Getting a Property*

```
4  IonaMessageContext context =
     (IonaMessageContext)contReg.getCurrent();

5  QName refName = new QName("http://widgets.com/","mySOAPHeader");
6  headerType header =
     (headerType)context.getRequestContext(refName);
...
}
```

The code in Example 148 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Gets a handle to the application's bus.
3. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
4. Calls `getCurrent()` on the context registry to get the message context for the application's thread and casts it to an Artix message context.
5. Creates the QName used to get the property from the context container. This QName must be the same QName as the one with which the property was registered.
6. Gets the customer SOAP header property by calling `getRequestContext()` with the appropriate `QName`.

Working with a property

Once you have gotten a property from the context container, you must first cast the returned `Object` to the appropriate data type for the property. Each property has its own associated data type. For example, in Example 148 the custom SOAP header's data is of type `headerType`.

Once the property is cast into the appropriate type you can access its fields using the methods defined for the type. Any changes made to the property by your application change the copy stored in the context container and will be propagated when the property is sent with a message.

Removing a property

If you do not want the data in a particular property to be propagated beyond a certain point, you can remove it from a context container using one of the the Artix message context's remove methods. `removeReplyContext()` removes properties from the message context's reply container and

`removeRequestContext()` removes properties from the message context's request context container. This is useful if your application must forward requests to other servers that do not need, or should not get, a property.

The removal methods take a single parameter, `name`, that specifies the `QName` of the property you are removing from the container. Example 149 shows the code for removing the HTTP client attributes from the request context container.

Example 149: *Removing a Property*

```
//Java
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
2 com.iona.jbus.Bus def_bus = DispatchLocals.getCurrentBus();

3 ContextRegistry contReg = def_bus.getContextRegistry();

4 IonaMessageContext context =
     (IonaMessageContext)contReg.getCurrent();

5 QName contextName = new QName("http://widgets.com/",
                                "HTTPClientAttributes");

6 context.removeRequestContext(contextName);
...
}
```

The code in Example 149 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Gets a handle to the application's bus.
3. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
4. Calls `getCurrent()` on the context registry to get the message context for the application's thread and casts it to an Artix message context.
5. Creates the QName for the property to remove.

6. Removes the HTTP client attribute property by calling `removeRequestContext()` with the appropriate `QName`.

Sending Header Information Using Contexts

Overview

Using the context mechanism, you can embed data in message headers that are not part of the operation's parameter list. This is useful in sending metadata such as security tokens or session information that is not vital to the logic involved in processing the request. Currently only SOAP headers are supported.

The data sent in the message header is a custom context that you will need to create and register with the Artix context container when you build your application.

Note: If you change the payload format used by the application, your code will continue to work. However, the header information stored in the context will not be transmitted.

To send customer header information in a context you need to do the following:

1. Define an XMLSchema for the data being stored in the header.
2. Generate the type factory and support code for the header data.
3. Register the type factory for the header data. See "Registering Type Factories" on page 128.
4. Register the header data as a context.

Once the header data is registered as a context with Artix, it can be accessed using the normal context mechanisms.

In this section

This section discusses the following topics:

Defining Context Data Types	page 219
Registering Context Types	page 221
SOAP Header Example	page 223

Defining Context Data Types

Overview

Contexts can store data of any XMLSchema type that is derived from `xsd:anyType`. In other words, a context data type can be any primitive, simple, or complex XMLSchema type.

When creating a context whose type is an XMLSchema primitive type or a native XMLSchema simple type like `xsd:nonNegativeInteger`, you do not need to explicitly define the context's data type. However, if you are creating a context whose type is a user-defined simple type or a complex type, you need to define the data type in an XMLSchema document (XSD) or in the types section of your contract and generate the appropriate type factories for the data type.

Defining a context schema

It is usually more appropriate to define a context data type (or types) in a separate schema file, rather than including the definition in the application's WSDL contract. This approach is more logical because contexts are normally used to implement features independently of any particular WSDL contract.

To define a complex context data type, *`ContextDataType`*, in the namespace, *`ContextDataURI`*, you define a context schema following the outline shown in Example 150.

Example 150:*Outline of a Context Schema*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="ContextDataURI"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified">
  <xsd:complexType name="ContextDataType">
        ...
  </xsd:complexType>
</xsd:schema>
```

Example

For example, you could define the data for a header that contains two elements. One element, `originator`, is a string containing the name of the message originator. The other element, `timeStamp`, is the date and time the message was sent. The data type for this header, `headerInfo`, is shown in Example 151.

Example 151: *Header Context Data Definition*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://schemas.iona.com/types/context"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified">
  <xsd:complexType name="headerInfo">
    <xsd:sequence>
      <xsd:element name="originator" type="xsd:string"/>
      <xsd:element name="timeStamp" type="xsd:dateTime"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Generating Java code for a context schema

To generate the Java code for the context data type, *`ContextType`*, from a context schema file, *`ContextSchema`*`.xsd`, enter the following command at the command line:

```
wsdltojava ContextSchema.xsd
```

The WSDL-to-Java compiler will generate two Java classes:

- *`ContextType`*`.java` contains the class representing the data type.
- *`ContextType`*`TypeFactory.java` contains the type factory needed to instantiate the context data type.

These classes will need to be accessible to any applications that wish to register and use a context of the defined type.

For more information on type factories see "Working with Artix Type Factories" on page 125.

Registering Context Types

Overview

Before you can use a context, you must register it with the bus' context registry using the registry's `registerContext()` method. `registerContext()` require that you provide the `QName` for the context and the `QName` of the data type stored in the context.

The main effect of registering a context is that the context registry adds a type factory reference to its internal table. This type factory reference enables the context registry to create context data instances whenever they are needed.

Registering a context to use as a SOAP Header

To register a context to be used as a SOAP header you need to provide the name of the WSDL message part that is to be inserted into the SOAP header. This information comes from the WSDL contract defining the messages used by the application.

Example 152 shows the signature of the `registerContext()` function used to register a context to be used as a SOAP header.

Example 152: *The registerContext() Function for SOAP Headers*

```
void ContextRegistry.regiserContext(QName name, QName type,
                                    QName message_name,
                                    String part_name);
```

`registerContext()` takes the following arguments:

`name`	The qualified name used to represent the property.
`type`	The qualified name of the property's data type.
`message_name`	The qualified name of the WSDL message specified in the `<soap:header>` element defining this SOAP header. If there is no `<soap:header>` elements defined in the contract, this can be any valid `QName`.
`part_name`	The part name specified in the `<soap:header>` element defining this SOAP header. If there is no `<soap:header>` elements defined in the contract, this can be any valid `String`.

For example, to register a SOAP header property of the type defined in Example 151 on page 220 you would use code similar to Example 153.

Example 153:*Registering a SOAP Header Property*

```
// Java

1    // Artix servant, servant, obtained earlier
     headerInfoTypeFactory fact = new headerInfoTypeFactory();
     servant.registerTypeFactory(fact);

2    // Bus, bus, obtained earlier
     ContextRegestry contReg = bus.get_context_registry();

3    // Create a QName for the new property
     QName name = new QName("http://javaExamples.iona.com",
                            "SOAPHeader");

4    // Create a QName to hold the QName of the property's data type
     QName type =  new QName("http://schemas.iona.com/types/context",
                             "headerInfo");

5    // Create a QName for the message
     QName message = new QName("http://myHeader.com/header"
                               "header_info");

6    // Register the property
     contReg.registerContext(name, type, message, "header_part");
```

The code in Example 153 does the following:

1. Register the type factory for the header's data type.
2. Get a handle to the bus' context registry.
3. Build the `QName` by for the new property. This can be any valid `QName`.
4. Build the `QName` that specifies the property's data type. The values for this are taken from the XSD defining the data type. The first argument is the namespace under which the type is defined. The second argument is the name of the complex type.
5. Build the `QName` for the message defining the SOAP header. In this example, the SOAP header is not defined in the WSDL contract so the value is unimportant.
6. Register the property with the context registry. The value used for the part name, `header_part`, can be any string.

SOAP Header Example

Overview

The example in this section transmits a custom SOAP header between two Artix processes. The SOAP header is defined in the WSDL contract for this example to demonstrate how context registration relates to the WSDL contract for SOAP headers.

The SOAP header data in this example is transmitted as follows:

1. The client registers the property, `SOAPHeaderInfo`, with the context registry for its bus.
2. The client initializes the property instance.
3. The client invokes the `sayHi()` operation on the server and the SOAP header property is packaged into the request message's SOAP header.
4. When the server starts up, it registers the `SOAPHeaderInfo` property with the context registry for its bus.
5. When the `sayHi()` operation request arrives on the server side, the SOAP header is extracted and put into the request context container as a `SOAPHeaderInfo` property.
6. The `sayHi()` operation implementation extracts the property from the request.

If the server is this example were not an Artix process, it would not need to use the context mechanism to extract the SOAP header. It would have its own method of handling the SOAP header.

WSDL contract

Example 154 on page 224 shows the WSDL contract used to define the service used in this example. It imports the XSD file, `SOAPcontext.xsd`, that defines the SOAP header property's data type in Example 151 on page 220. The `SOAPHeaderInfo` type is used to define the only part of the `headerMsg` message. In the SOAP binding for `Greeter`, `GreeterSOAPBinding`, the definition of the input message includes a `<soap:header>` element that

specifies that `headerMsg:headerPart` is to be placed in a SOAP header when a request is made. Your application code will be responsible for creating the property that populates the defined SOAP header.

Example 154:*SOAP Header WSDL*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorld" targetNamespace="http://www.iona.com/soapHeader"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:ns1="http://schemas.iona.com/types/context"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <import location="file:/SOAPcontext.xsd"
          namespace="http://schemas.iona.com/types/context"/>
  <types>
    <schema targetNamespace="http://www.iona.com/custom_soap_header"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="responseType" type="xsd:string"/>
      <element name="requestType" type="xsd:string"/>
    </schema>
  </types>
  <message name="greetMeRequest">
    <part element="requestType" name="me"/>
  </message>
  <message name="greetMeResponse">
    <part element="responseType" name="theResponse"/>
  </message>
  <message name="headerMsg">
    <part type="ns1:SOAPHeaderInfo" name="headerPart" />
  </message>
  <portType name="Greeter">
    <operation name="greetMe">
      <input message="greetMeRequest" name="greetMeRequest"/>
      <output message="greetMeResponse" name="greetMeResponse"/>
    </operation>
  </portType>
```

Example 154: *SOAP Header WSDL*

```
  <binding name="GreeterSOAPBinding" type="Greeter">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="greetMe">
       <soap:operation soapAction="" style="document"/>
       <input name="greetMeRequest">
         <soap:body use="literal"/>
         <soap:header use="literal" message="headerMsg" part="headerPart" />
       </input>
       <output name="greetMeResponse">
         <soap:body use="literal"/>
       </output>
    </operation>
  </binding>
  <service name="GreeterService">
    <port binding="GreeterSOAPBinding" name="SoapPort">
      <address location="http://localhost:9000"/>
    </port>
  </service>
</definitions>
```

Generating the Java classes for the property's data type

Because the XSD file for the property's data type is imported into the contract for the service `wsdltojava` will automatically generate the appropriate classes and type factories for `SOAPHeaderInfo` when you generate the code for the service.

To generate the code for the service save the WSDL contract into a file called `soapHeader.wsdl` and the XSD file for `SOAPHeaderInfo` into a file called `SOAPcontext.xsd`. Then run the following command:

```
wsdltojava soapHeader.wsdl
```

The client

The client in this example will send a SOAP header of type `SOAPHeaderInfo` when it invokes the `greetMe` operation. To do this it must do four things:

1. Register the type factory for `SOAPHeaderInfo`.
2. Register a property of `SOAPHeaderInfo` type.
3. Create an instance of `SOAPHeaderInfo`.
4. Populate the instance with the appropriate data.
5. Set the `SOAPHeaderInfo` property in the request context container.

When the `greetMe()` method is invoked, the property will be inserted into the SOAP message's header element and sent to the server.

Example 155 on page 226 shows the code for the client.

Example 155: *Client Code*

```
// Java
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;

public class GreeterClient
{

  public static void main (String args[]) throws Exception
  {
1   Bus bus = Bus.init(args);

2   QName name = new QName("http://www.iona.com/soapHeader",
                           "GreeterService");
    QName portName = new QName("","Greeter");

    String wsdlPath = "file:/./soapHeader.wsdl";
    URL wsdlLocation = new File(wsdlPath).toURL();

    ServiceFactory factory = ServiceFactory.newInstance();

    Service service = factory.createService(wsdlLocation, name);

    Greeter impl = (Greeter)service.getPort(portName,
                                            Greeter.class);

3   SOAPHeaderInfoTypeFactory fact =
      new SOAPHeaderInfoTypeFactory();
    bus.registerTypeFactory(fact);

4   ContextRegestry contReg = bus.getContextRegistry();
```

Example 155:*Client Code*

```
5       QName name = new QName("http://javaExamples.iona.com",
                               "SOAPHeader");

6       QName type =
          new QName("http://schemas.iona.com/types/context",
                    "SOAPHeaderInfo");

7       QName message = new QName("http://www.iona.com/soapHeader"
                                  "headerMsg");

8       contReg.registerContext(name, type, message, "headerPart");

9       SOAPHeaderInfo header = new SOAPHeaderInfo();
        header.setOriginator("IONA Technologies");
        header.setMessage("Artix is powerful!");

10      IonaMessageContext context =
          (IonaMessageContext)contReg.getCurrent();

11      context.setRequextContext(name, header);

12      String string_out;

        string_out = impl.greetMe("Chris");
        System.out.println(string_out);

        bus.shutdown(true);
        }
}
```

The code in Example 155 on page 226 does the following:

1. Initializes an instance of the bus.
2. Creates a proxy for the Greeter service.
3. Register the type factory for `SOAPHeaderInfo`.
4. Gets the context registry from the bus.
5. Builds the `QName` for the new property.
6. Builds the `QName` for the property's data type. The values for this are taken from the XSD defining the data type. The first argument is the namespace under which the type is defined. The second argument is the name of the complex type.

7. Builds the `QName` for the message defining the SOAP header. In this example, the SOAP header is in the WSDL contract so the value is the `QName` for the message defined in the `<soap:header>` element of the contract, `http://www.iona.com/soapHeader:headerMsg`.
8. Registers the property with the context registry. The value used for the part name, `headerPart`, is the part name specified in the contract's `<soap:header>` element.
9. Instantiates an instance of the SOAP header property's class, `SOAPHeaderInfo`, and sets the fields.
10. Gets the Artix message context for the client.
11. Adds the SOAP header property to the request context container.
12. Invokes `greetMe()`. The SOAP header property is placed into the SOAP header of the request and sent to the server.

The server main line

The server must also register the `SOAPHeader` property with its context registry in order to extract the SOAP header sent with the request. Because the property only needs to be registered with the context registry once, it makes sense to register it in the server main line before control is passed to the bus.

Example 156 on page 228 shows the code for the server's main line.

Example 156: *Server main()*

```
// Java
import com.iona.jbus.*;
import javax.xml.namespace.QName;

public class Server
{
  public static void main(String args[])
  throws Exception
  {
1   // Initialize the Artix bus
    Bus bus = Bus.init(args);
```

Example 156: *Server main()*

```
2     // Register the implementation object factory
      QName name = new QName("http://www.iona.com/soapHeader",
                             "GreeterService");
      Servant servant =
                   new SingleInstanceServant("./soapHeader.wsdl",
                                             new GreeterImpl());
      bus.registerServant(servant, name, "SoapPort");

3     SOAPHeaderInfoTypeFactory fact =
        new SOAPHeaderInfoTypeFactory();
      bus.registerTypeFactory(fact);

4     ContextRegestry contReg = bus.getContextRegistry();

5     QName propName =  new QName("http://javaExamples.iona.com",
                                  "SOAPHeader");

6     QName propType =
                new QName("http://schemas.iona.com/types/context",
                          "SOAPHeaderInfo");

7     QName message = new QName("http://www.iona.com/soapHeader"
                                "headerMsg");

8     contReg.registeContext(propName, propType,
                             message, "headerPart");

9     // Start the Bus
      bus.run();
    }
}
```

The code in Example 156 on page 228 does the following:

1. Initializes an instance of the bus.
2. Registers the services implementation object with the bus.
3. Registers the type factory for `SOAPHeaderInfo`.
4. Gets the context registry from the bus.
5. Builds the `QName` for the new property.

6. Builds the `QName` for the property's data type. The values for this are taken from the XSD defining the data type. The first argument is the namespace under which the type is defined. The second argument is the name of the complex type.
7. Builds the `QName` for the message defining the SOAP header. In this example, the SOAP header is in the WSDL contract so the value is the `QName` for the message defined in the `<soap:header>` element of the contract, `http://www.iona.com/soapHeader:headerMsg`.
8. Registers the property with the context registry. The value used for the part name, `headerPart`, is the part name specified in the contract's `<soap:header>` element.
9. Hands control over to the bus.

The implementation object

The service's implementation object, `GreeterImpl`, gets the SOAP header from the request message and prints the headers contents. To do this the implementation object must get the SOAP header property from the request context container. Getting the SOAP header property takes four steps:

1. Get a reference to the bus for the implementation object.
2. Get the bus' context registry.
3. Get the thread's Artix message context from the registry.
4. Get the SOAP header property from the request context container.

Example 157 shows the code for the `GreeterImpl` implementation object.

Example 157: *Implementation of the Greeter Service*

```
// Java
import java.net.*;
import java.rmi.*;
import javax.xml.namespace.QName;

import com.iona.jbus.*

public class GreeterImpl
{
  public String greetMe(String stringParam)
  {
1 com.iona.jbus.Bus bus = DispatchLocals.getCurrentBus();
```

Example 157: *Implementation of the Greeter Service*

```
2    ContextRegestry contReg = bus.getContextRegistry();

3    IonaMessageContext context =
       (IonaMessageContext)contReg.getCurrent();

4    QName name =  new QName("http://javaExamples.iona.com",
                             "SOAPHeader");

5    SOAPHeaderInfo header = (SOAPHederInfo)
       reqContext.getRequestContext(name);

6    System.out.println("SOAP Header Originator:
                        "+header.getOriginator());
     System.out.println("SOAP Header message:
                        "+header.getMessage());

7    return "Hello Artix User: "+stringParam;
     }
}
```

The code in Example 157 on page 230 does the following:

1. Gets an instance of the bus.
2. Gets the context registry from the bus.
3. Gets the context current for the implementation object's thread.
4. Builds the `QName` for the SOAP header property. This `QName` must be the same as the `QName` used when registering the property in the server main.
5. Gets the SOAP header property from the request context container.
6. Prints out the information contained in the SOAP header.
7. Returns the results of the operation to the client.

CHAPTER 11

Developing Java Plug-Ins

Java Plug-Ins can perform a number of tasks including registering servants or implementing message handlers.

Overview

Developing and loading an Artix plug-in in Java requires you to perform four tasks:

1. Extend the `BusPlugIn` class to implement your plug-in's application logic.
2. Implement the `BusPlugInFactory` interface.
3. If needed, configure the plug-in's class loader environment. See "Class Loading" on page 41.
4. Configure Artix to use the plug-in. See *Deploying and Managing Artix Solutions*.

In this chapter

This chapter discusses the following topics:

Extending the BusPlugIn Class	page 234
Implementing the BusPlugInFactory Interface	page 237

Extending the BusPlugIn Class

Overview

The `BusPlugIn` class is the base class for all Artix plug-ins. It provides a method, `getBus()`, that returns the bus with which the plug-in is associated. In addition, it has two abstract classes that you must implement:

- A constructor for your class.
- The `busInit()` method called by the bus to initialize the plug-in.
- The `busShutdown()` method called by the bus when it is shutting down to allow the plug-in to perform any clean-up it needs before being destroyed.

Implementing the constructor

The constructor for your plug-in has two requirements:

1. Its first argument must be a bus instance.
2. It must call `super()` with the passed in bus reference.

Example 158 shows a constructor for a plug-in called `BankPlugIn`. It simply calls `super()` on the bus instance. It could, however, have performed some logging operations or initialized resources.

Example 158: *BusPlugIn constructor*

```
// Java
public class BankPlugIn extends BusPlugIn
{
  public BankPlugIn(Bus bus)
  {
    super(bus);
  }
...
}
```

busInit()

`busInit()` is called by every bus that loads your plug-in. Inside `busInit()`, you perform all of the initialization needed for your plug-in to perform its job. For example, if your plug-in implemented a service defined in WSDL you

would create and register the servant in `busInit()`. If your plug-in implemented a JAX-RPC Handler, you would register your handler factory in `busInit()`.

Example 159 shows a `busInit()` method used in implementing the bank service, described in *Developing Artix Applications in Java*, as a plug-in.

Example 159: *busInit()*

```
// Java
import com.iona.jbus.*;
import com.iona.jbus.servants.*;
import javax.xml.namespace.QName;

import java.net.*;
import java.io.*;

public class BankPlugIn extends BusPlugIn
{
 private BankImpl bank;
 ...
  public void busInit() throws BusException
  {
    Bus bus = getBus();
    QName qname = new QName("http://www.iona.com/bus/demos/bank",
                            "BankService");
    bank = new BankImpl();
    Servant servant = new SingleInstanceServant(bank,
                            "./bank.wsdl", bus);
    bus.registerServant(servant, qname, "BankPort");
  }

  ...
}
```

busShutdown()

`busShutdown()` is called on the plug-in by the bus when the bus is shutting down. Once `busShutdown()` completes, the bus calls `destrotBusPlugIn()` on the plug-in factory object. This is good place to release instance specific resources used by the plug-in or to do other house keeping. For example, the bank plug-in may need to force the account objects it created to finish any running transactions and flush their information to the permanent store before shutting down as shown as shown in Example 160.

Example 160:*busShutdown()*

```
// Java
import com.iona.jbus.*;
import com.iona.jbus.servants.*;
import com.iona.schemas.references.Reference;

import javax.xml.namespace.QName;
import java.net.*;
import java.io.*;

public class BankPlugIn extends BusPlugIn
{
  private BankImpl bank;
  ...
  public void busShutdown() throws BusException
  {
    Account acctProxy;
    Reference ref;
    Bus bus = getBus()
    Iterator it = bank.accounts.values().interator();

    while(it.hasNext())
    {
      ref = (Reference)it.next();
      acctProxy = bus.createClient(ref, Account.class);
      acctProxy.closeDown();
    }
  }
}
```

Implementing the BusPlugInFactory Interface

Overview

The `BusPlugInFactory` interface provides the methods used by the Artix bus to manage a plug-in implementation. It has two methods you must implement:

- `createBusPlugIn()` creates instances of the plug-in and its associated resources and associate them with particular bus instances.
- `destroyBusPlugIn()` destorys plug-in instances and frees the resources associated with them.

createBusPlugIn()

`createBusPlugIn()` is called by a bus instance when it loads a plug-in. In most instances, `createBusPlugIn()` will simply instaniate an instance of your plug-in object and return it. However, you can use this method to initialize any global resources used by the plug-in.

Example 161 shows the signature for `createBusPlugIn()`.

Example 161: *createBusPlugIn()*

```
public BusPlugIn createBusPlugIn(Bus bus) throws BusException;
```

destroyBusPlugIn()

`destroyBusPlugIn()` is called by a bus instance when it is shutting down and releasing its resources. In most instances, this method does not need to do anything. However, if you created any global resources for your plug-in this would be a convinient place to free them.

Example 162 shows the signature for `destroyBusPlugIn()`.

Example 162: *destroyBusPlugIn()*

```
public void destroyBusPlugIn(BusPlugIn plugin);
```

Example

For example, the `BusPlugInFactory` implementation for a plug-in `BankPlugIn` would look similar to Example 163.

Example 163: *BankPlugInFactory*

```
// Java
import com.iona.jbus.*;

public class BankPlugInFactory implements BusPlugInFactory
{
  public BusPlugIn createBusPlugIn(Bus bus) throws BusException
  {
    return new BankPlugIn(bus);
  }

  public void destroyBusPlugIn(BusPlugIn plugin)
  throws BusException
  {
  }
}
```

CHAPTER 12

Writing Message Handlers

Using the JAX-RPC Handler mechanism, developers can access and manipulate messages as they pass along the delivery chain.

In this chapter

This chapter discusses the following topics:

Message Handlers: An Introduction	page 240
Developing Request-Level Handlers	page 244
Developing Message-Level Handlers	page 251

Message Handlers: An Introduction

Overview

When a service proxy invokes an operation on a service, the operations parameters are passed to the Artix bus where they are built into a message and placed on the wire. When the message is received by the service, the Artix bus reads the message from the wire, reconstructs the message, and then passes the operation parameters to the application code responsible for implementing the operation. When the service is finished processing the request, the reply message undergoes a similar chain of events on its trip to the server. This is shown in Figure 9.

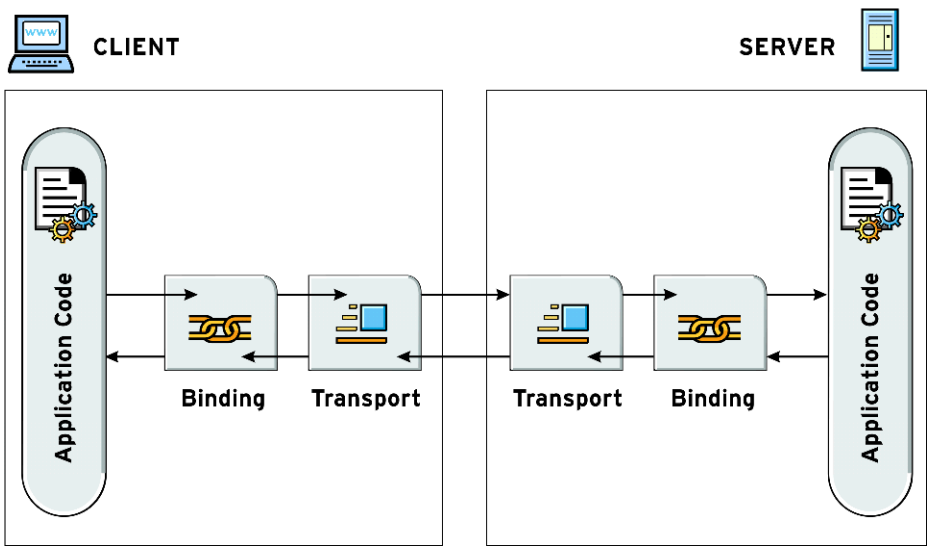

Figure 9: *The Life of a Message*

You can write message handlers that work with a message at each stop along its path. For example, if you wanted to compress a message before sending it on the wire, you could write a message handler that takes the message data from the binding and compresses it before the transport puts

the message on the wire. Likewise, you could write a message handler that takes the message from the transport and decompresses it before passing it on to the binding.

Message handler levels

The JAX-RPC specification outlines a mechanism for developers to write custom message handlers using the `Handler` interface. Using the handler mechanism, you can intercept and work with message data at four points along the request message's life cycle and at four points along the reply message's life cycle. Both requests and replies can be handled at the client request level, the client message level, the server message level, and the server request level. These levels are shown in Figure 10.

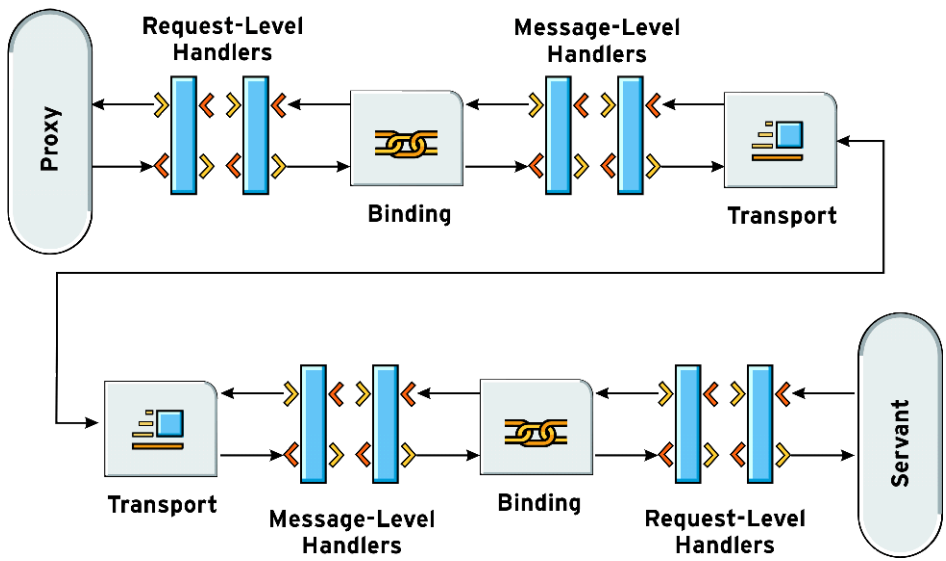

Figure 10: *Handler Levels*

On the client side of an application, you can write message handlers to process requests as they pass from the application to the binding and to process responses as they passes from the binding to the application. These are called *request-level handlers*. You can also write message handlers to

process requests as they pass from the binding to the transport and to process responses as they pass from the transport to the binding. These are called *message-level handlers*.

On the server side of an application the direction of the message flow is reversed, but the levels stay the same. For example, a request-level handler on the server side would work with requests as they pass from the binding to the application and a message-level handler would process with responses as they passed from the binding to the transport.

Implementing a message handler

Message handlers are developed as Artix plug-ins. This allows you to develop a message handler once an reuse it in any Artix Java application. Writing a plug-in requires that you implement the `BusPlugInFactory` interface and extend the `BusPlugIn` class to initialize the message handlers. For details on the plug-in interfaces see "Developing Java Plug-Ins" on page 233.

To write a message handler, you implement the JAX-RPC `Handler` interface and the `HandlerFactory` interface. To make implementing these interfaces easier, Artix supplies a `GenericHandler` class and a `GenericHandlerFactory` class that you can extend to write your handlers. These generic classes provide idle implementations of all of the methods for the interfaces. By extending them you only to provide implementations for the methods needed by your message handler.

Your `Handler` implementation contains the logic for the message handler you are writing. The `Handler` interface has two methods that process messages: `handleRequest()` and `handleResponse()`. `handleRequest()` is invoked when a request message is passing through the handler. `handleResponse()` in invoked when a response message is passing through the handler. These methods are invoked in both request level handlers and message level handlers.

A `HandlerFactory` implementation is responsible for instantiating bus specific instances of one or more message handlers. The `HandlerFactory` interface has four methods for instantiating handlers: `getClientMessageHandler()`, `getClientRequestHandler()`, `getServerMessageHandler()`, and `getServerRequestHandler()`. As the method names imply, each method is used to instantiate a message handler for use at a specific point in the messaging chain. For example, `getClientMessageHandler()` would be called by the bus to instantiate a client side message handler for processing messages as the passed between

the binding and the transport. Each method in a factory can instantiate one message handler. However, a factory can be developed to instantiate four message handlers because the bus will only call the factory method needed to instantiate the message handler configured to be used at a particular point in the message chain.

Configuring Artix to use message handlers

Before your applications can use message handlers, you must configure them to load the message handlers at the appropriate points in the message chain. This is done by adding the following configuration variables into the application's configuration scope:

binding:artix:client_message_interceptor_list is an ordered list of QNames specifying the message-level handlers for a client.

binding:artix:client_request_interceptor_list is an ordered list of QNames specifying the request-level handlers for a client.

binding:artix:server_message_interceptor_list is an ordered list of QNames specifying the message-level handlers for a server.

binding:artix:server_request_interceptor_list is an ordered list of QNames specifying the request-level handlers for a server.

The message handlers are placed in the list in the order they will be invoked on the message as it passes through the messaging chain. For example, if the server request interceptor list was specified as `"tns:Freeze+tns:Dry"`, a message would be passed into the message handler `Freeze` as it left the binding. Once Freeze processed the message, it would be passed into `Dry` for more processing. `Dry` would then pass the message along to the application code. For more information on configuring Artix applications see *Deploying and Managing Artix Applications*.

Developing Request-Level Handlers

Overview

Request-level handlers process messages as they pass between your application code and the binding that formats the message that is being sent on the wire. On the client side, request messages are processed immediately after the application invokes a remote method on its service proxy and before the binding formats the message. Responses are processed after the message is decoded by the binding and before the data is returned to the client application code. On the server side, requests are processed as they pass from the binding to the service implementation. Replies are processed as they pass from the server implementation to the binding.

Currently, message handlers at the request level have limited access to the message data. They can access the applications message context, access the messages SOAP headers, or access the messages security properties. For example, your application could have a client side message handler that added a custom SOAP header to its requests for authorization purposes. The server could then use a message handler to read the SOAP header and perform the authorization before the request gets to the service implementation.

Procedure

To develop a request-level handler you need to do the following:

1. Implement a `BusPluginFactory` to load the plug-in that implements your message handler. See "Implementing the BusPlugInFactory Interface" on page 237.
2. Extend `BusPlugin` to load your message handler.
3. Implement a `HandlerFactory` to instantiate your message handler when the bus needs it.
4. Implement a `Handler` to host the logic used by your message handler.
5. Configure your application to load the message handler plug-in.
6. Configure your application to include the message handler in the request handler chain. See *Deploying and Managing Artix Solutions*.

The plug-in

Your implementation of `busInit()` in your plug-in must register the handler factory used to instantiate your message handler. Handler factory registration is done using the bus' `registerHandlerFactory()` method. The signature for `registerHandlerFactory()` is shown in Example 164.

Example 164: *registerHandlerFactory()*

```
void registerHandlerFactory(HandlerFactory factory);
```

`registerHandlerFactory()` takes an instance of the handler factory for your message handler. Subsequent calls to `registerHandlerFactory()` add to the list of registered handler factories. So, if you need to register multiple handler factories you simply call `registerHandlerFactory()` with an instance of each handler factory to be registered.

Example 165 shows a the plug-in code for a message handler.

Example 165: *Message Handler Plug-In*

```
//Java

import com.iona.jbus.*;

public class HandlerPlugIn extends BusPlugIn
{
  public HandlerPlugin(Bus bus)
  {
    super(bus);
  }

  public void busInit() throws BusException
  {
    try
    {
      Bus bus = getBus();

      bus.registerHandlerFactory(new emoHandlerFactory());
    }
    catch (Exception ex)
    {
      throw new BusException(ex);
    }
  }
```

Example 165:*Message Handler Plug-In*

```
  public void busShutdown() throws BusException
  {
  }
}
```

The code in Example 165 does the following:

1. Imports the Artix bus APIs.
2. Implements a constructor for the plug-in class.
3. Implements `busInit()` to register the handler factory.
4. Gets the plug-in's bus.
5. Registers the message handler's factory with the bus using `registerHandlerFactory()`.
6. Implements `busShutdown()`.

The handler factory

The easiest way to develop your handler factory is to extend the `GenericHandlerFactory` included with Artix. You can also implement the standard JAX-RPC `HandlerFactory` interface. The `GenericHandlerFactory` implements all of the methods in the `HandlerFactory` interface, so you only need to override the methods needed for your message handlers and provide a constructor for your handler factory.

When developing request-level handlers, the two handler factory methods that are of interest are `getClientRequestHandler()` and `getServerRequestHandler()`. As their names imply they are used to instantiate request-level handlers for either a client or a server. Depending on your message handler implementation, you can override one or both of these methods. For example, you could develop a single handler factory for both the client side and the server side request-level handlers. The bus will call the appropriate method to instantiate the correct handler.

The signatures for `getClientRequestHandler()` and `getServerRequestHandler()` are shown in Example 166. They take no arguments and return an instance of the class `HandlerInfo`.

Example 166:*Handler Factory Methods for Request Level Handlers*

```
public javax.xml.rpc.handler.HandlerInfo getClientRequestHandler()
public javax.xml.rpc.handler.HandlerInfo getServerRequestHandler()
```

The returned `HandlerInfo` object needs to contain all the information needed by the bus to manage your message handler. You need to supply the `Class` that implements your message handler. For example if your client side message handler is implemented by a class called `emoClientRequestHandler`, you need to set the returned HandlerInfo's `HanderClass` field to `emoClientRequestHandler.class` by invoking `setHandlerClass()` on the `HandlerInfo` object.

Example 167 shows code for implementing a handler factory.

Example 167: *Handler Factory For Request Level Handlers*

```
//Java
import com.iona.jbus.*;
import com.iona.jbus.servants.*;
import javax.xml.namespace.QName;

import java.net.*;
import java.io.*;

import javax.xml.rpc.handler.*;

1  public class TestHandlerFactory extends GenericHandlerFactory
   {
2    public TestHandlerFactory()
     {
       super(new QName("http://www.iona.com/bus/tests",
                       "TestInterceptor"));
     }

3    public HandlerInfo getClientRequestHandler()
     {
4      HandlerInfo info = new HandlerInfo();
5      info.setHandlerClass(emoClientRequestHandler.class);
       return info;
     }

     public HandlerInfo getServerRequestHandler()
     {
       HandlerInfo info = new HandlerInfo();
       info.setHandlerClass(emoServerRequestHandler.class);
       return info;
     }
   }
```

The code in Example 167 does the following:

1. Extends `GenericHandlerFactory`.
2. Implements a constructor for the handler factory. The QName set is the QName used by the bus to reference the handler factory.
3. Overrides `getClientRequestHandler()`.
4. Instantiates a `HandlerInfo` object.
5. Sets the `HandlerClass` property to the class of the message handler that will process client requests.

The handler

The easiest way to develop your message handler logic is to extend the `GenericHandler` class supplied with Artix. The `GenericHandler` class provides implementations for all of the methods in the JAX-RPC `Handler` interface, so all you need to do is override the methods your message handler requires. You can also implement the JAX-RPC `Handler` interface if you desire.

The `Handler` interface has two methods that are used to process messages: `handleRequest()` and `handleResponse()`. `handleRequest()` processes request messages and `handleResponse()` processes reply messages. The bus will call these methods at the appropriate place in the messaging chain to process the message data. It is important to remember where in the messaging chain the message handler is called. For example, a message handler that reads a SOAP header from a request in the server will not work if it is placed in the client request chain.

The signatures for `handleRequest()` and `handleResponse()` are shown in Example 168. Both methods have a `MessageContext` as an argument. For information on using the message contexts see "Using Message Contexts" on page 197. The return value should reflect the state of the message processing. If the message is successfully processed return `true`. If not, return `false`.

Example 168: *handleRequest() and handleResponse()*

```
boolean handleRequest(MessageContext context);
boolean handleResponse(MessageContext context);
```

At the request level, your message handler can access both the generic message context and the Artix specific context. Because the properties of the generic message context do not effect the message as it passes through

the messaging chain, it is more likely that your message handler will use the Artix specific message context. Properties set into the Artix specific message context at the request-level will be propagated down the message chain and effect how the message is formatted and transmitted. For example, security properties and SOAP headers manipulated in a client request handler will change the properties that are sent to the server. On the return side of the messaging chain, such as in a server request handler or a client response handler, the request-level is the level in which the SOAP header and security properties are made available.

Example 169 shows the code for a client request-level message handler that sets a SOAP header on the request and reads the SOAP header returned with the response.

Example 169:*Client Request Level Message Handler*

```
// Java
import com.iona.jbus.IonaMessageContext;
import com.iona.jbus.ContextException;

import javax.xml.namespace.QName;

import javax.xml.rpc.handler.*;

public class emoClientRequestHandler extends GenericHandler
{
  public boolean handleRequest(MessageContext context)
  {
    IonaMessageContext mycontext = (IonaMessageContext)context;
    QName principalCtxName = new QName("", "SOAPHeaderInfo");
    SOAPHeaderInfo requestInfo = new SOAPHeaderInfo();
    requestInfo.setOriginator("Client");
    requestInfo.setMessage("Hello from Client!");
    mycontext.setRequestContext(principalCtxName,requestInfo);

    return true;
  }
```

Example 169:*Client Request Level Message Handler*

```
  public boolean handleResponse(MessageContext context)
  {
    IonaMessageContext mycontext = (IonaMessageContext)context;
    QName ctxName = new QName("", "SOAPHeaderInfo");
    SOAPHeaderInfo replyInfo =
   (SOAPHeaderInfo)mycontext.getReplyContext(ctxName);
    System.out.println("Header from Server: ");
    System.out.println("Originator - " +
   replyInfo.getOriginator());
    System.out.println("Message    - " + replyInfo.getMessage());

    return true;
  }
}
```

Developing Message-Level Handlers

Overview

Message-level handlers process messages as they pass between the binding and the transport. On the client side, request messages are processed after the binding formats the message and before the transport writes it to the wire. Responses are processed after the message is read off of the wire and before it is decoded by the binding. On the server side, requests are processed after the message is read off of the wire and before it is decoded by the binding. Replies are processed as they pass from the binding to the transport.

Message handlers at the message level have access to the raw message stream that is being written out the wire. This data has been formatted into the appropriate message type specified by the binding. Message-level handlers can also access the applications message context. For example, your application could have a client side message handler that compresses the message data to enhance network performance. The server could then use a message handler to decompress the message data before it is sent to the binding for decoding.

Procedure

To develop a message-level message handler you need to do the following:

1. Implement a `BusPluginFactory` to load the plug-in that implements your message handler. See "Implementing the BusPlugInFactory Interface" on page 237.
2. Extend `BusPlugin` to load your message handler.
3. Implement a `HandlerFactory` to instantiate your message handler when the bus needs it.
4. Implement a `Handler` to host the logic used by your message handler.
5. Configure your application to load the message handler plug-in.
6. Configure your application to include the message handler in the message handler chain. See *Deploying and Managing Artix Solutions*.

The plug-in

Your implementation of `busInit()` in your plug-in must register the handler factory used to instantiate your message handler. Handler factory registration is done using the bus' `registerHandlerFactory()` method. The signature for `registerHandlerFactory()` is shown in Example 170.

Example 170: *registerHandlerFactory()*

```
void registerHandlerFactory(HandlerFactory factory);
```

`registerHandlerFactory()` takes an instance of the handler factory for your message handler. Subsequent calls to `registerHandlerFactory()` add to the list of registered handler factories. So, if you need to register multiple handler factories you simply call `registerHandlerFactory()` with an instance of each handler factory to be registered.

Example 171 shows a the plug-in code for a message handler.

Example 171: *Message Handler Plug-In*

```
//Java

1  import com.iona.jbus.*;

   public class HandlerPlugIn extends BusPlugIn
   {
2    public HandlerPlugin(Bus bus)
     {
       super(bus);
     }

3    public void busInit() throws BusException
     {
       try
       {
4        Bus bus = getBus();

5        bus.registerHandlerFactory(new emoHandlerFactory());
       }
       catch (Exception ex)
       {
         throw new BusException(ex);
       }
     }
```

Example 171: *Message Handler Plug-In*

```
6     public void busShutdown() throws BusException
      {
      }
}
```

The code in Example 171 does the following:

1. Imports the Artix bus APIs.
2. Implements a constructor for the plug-in class.
3. Implements `busInit()` to register the handler factory.
4. Gets the plug-in's bus.
5. Registers the message handler's factory with the bus using `registerHandlerFactory()`.
6. Implements `busShutdown()`.

The handler factory

The easiest way to develop your handler factory is to extend the `GenericHandlerFactory` included with Artix. You can also implement the standard JAX-RPC `HandlerFactory` interface. The `GenericHandlerFactory` implements all of the methods in the `HandlerFactory` interface, so you only need to override the methods needed for your message handlers and provide a constructor for your handler factory.

When developing message-level handlers, the two handler factory methods that are of interest are `getClientMessageHandler()` and `getServerMessageHandler()`. As their names imply they are used to instantiate message-level handlers for either a client or a server. Depending on your message handler implementation, you can override one or both of these methods. For example, you could develop a single handler factory for both the client side and the server side message-level handlers. The bus will call the appropriate method to instantiate the correct handler.

The signatures for `getClientMessageHandler()` and `getServerMessageHandler()` are shown in Example 172. They take no arguments and return an instance of the class `HandlerInfo`.

Example 172: *Handler Factory Methods for Message Level Handlers*

```
public javax.xml.rpc.handler.HandlerInfo getClientMessageHandler()
public javax.xml.rpc.handler.HandlerInfo getServerMessageHandler()
```

The returned `HandlerInfo` object needs to contain all the information needed by the bus to manage your message handler. You need to supply the `Class` that implements your message handler. For example if your client side message handler is implemented by a class called `emoClientMessageHandler`, you need to set the returned HandlerInfo's `HanderClass` field to `emoClientMessageHandler.class` by invoking `setHandlerClass()` on the `HandlerInfo` object.

Example 173 shows code for implementing a handler factory.

Example 173: *Handler Factory For Message Level Handlers*

```
//Java
import com.iona.jbus.*;
import com.iona.jbus.servants.*;
import javax.xml.namespace.QName;

import java.net.*;
import java.io.*;

import javax.xml.rpc.handler.*;

1  public class TestHandlerFactory extends GenericHandlerFactory
   {
2    public TestHandlerFactory()
     {
       super(new QName("http://www.iona.com/bus/tests",
                       "TestInterceptor"));
     }

3    public HandlerInfo getClientMessageHandler()
     {
4      HandlerInfo info = new HandlerInfo();
5      info.setHandlerClass(emoClientMessageHandler.class);
       return info;
     }

     public HandlerInfo getServerMessageHandler()
     {
       HandlerInfo info = new HandlerInfo();
       info.setHandlerClass(emoServerMessageHandler.class);
       return info;
     }
   }
```

The code in Example 173 does the following:

1. Extends `GenericHandlerFactory`.
2. Implements a constructor for the handler factory. The QName set is the QName used by the bus to reference the handler factory.
3. Overrides `getClientMessaageHandler()`.
4. Instantiates a `HandlerInfo` object.
5. Sets the `HandlerClass` property to the class of the message handler that will process client requests.

The handler

The easiest way to develop your message handler logic is to extend the `GenericHandler` class supplied with Artix. The `GenericHandler` class provides implementations for all of the methods in the JAX-RPC `Handler` interface, so all you need to do is override the methods your message handler requires. You can also implement the JAX-RPC `Handler` interface if you desire.

The `Handler` interface has two methods that are used to process messages: `handleRequest()` and `handleResponse()`. `handleRequest()` processes request messages and `handleResponse()` processes reply messages. The bus will call these methods at the appropriate place in the messaging chain to process the message data. It is important to remember where in the messaging chain the message handler is called. For example, a message handler that compresses a request in the client will cause unpredictable results if it is placed in the server message chain.

The signatures for `handleRequest()` and `handleResponse()` are shown in Example 174. Both methods have a `MessageContext` as an argument. For information on using the message contexts see "Using Message Contexts" on page 197. The return value should reflect the state of the message processing. If the message is successfully processed return `true`. If not, return `false`.

Example 174: *handleRequest() and handleResponse()*

```
boolean handleRequest(MessageContext context);
boolean handleResponse(MessageContext context);
```

At the message level, your message handler can access both the generic message context and a special `StreamMessageContext` that provides access to the raw message data that will be written onto the wire. Because the

properties of the generic message context do not effect the message as it passes through the messaging chain, it is more likely that your message-level handlers will use the raw message data. To get a `StreamMessageContext` you cast the `MessageContext` passed into the message handler method as shown in Example 175.

Example 175:*Getting a StreamMessageContext*

```
// Java
boolean handleResponse(MessageContext context)
{
  StreamMessageContext myCtx = (StreamMessageContext)context;
  ...
}
```

The `StreamMessageContext` has methods for getting and setting the input and output streams used by the transport as shown in Example 176. While `StreamMessageContext` provides methods for getting the output stream, you should always work with the input stream provided. Artix will ensure that data from the input stream is the data that gets propagated through the message chain.

Example 176:*StreamMessageContext*

```
package com.iona.jbus;

import javax.xml.rpc.handler.MessageContext;
import java.io.InputStream;
import java.io.OutputStream;

public interface StreamMessageContext extends MessageContext
{
  public static final String INPUT_STREAM_PROPERTY =
   "StreamMessageContext.InputStream";
  public static final String OUTPUT_STREAM_PROPERTY =
   "StreamMessageContext.OutputStream";

  public InputStream getInputStream();
  public void setInputStream(InputStream ins);
  public OutputStream getOutputStream();
  public void setOutputStream(OutputStream out);
}
```

Example 177 shows the code for a client message-level message handler that adds a string onto the end of a SOAP request before sending it to the server and removes an additional string from the end of the SOAP response before passing the SOAP message to the binding. The complete code for this demo can be found in the custom interceptor demo included in your Artix installation. `TestInputStream` extends `InputStream` to allow for adding a string to the end of the stream.

Example 177:*Client Message Level Message Handler*

```
// Java
import com.iona.jbus.*;

import java.io.*;
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.*;

public class emoClientMessageHandler extends GenericHandler
{
  public boolean handleRequest(MessageContext context)
  {
    StreamMessageContext smc = (StreamMessageContext)context;
    InputStream ins = smc.getInputStream();
    ins = new TestInputStream(ins,
                              TestInputStream.CLIENT_TO_SERVER);
    smc.setInputStream(ins);
    return true;
  }

  public boolean handleResponse(MessageContext context)
  {
    StreamMessageContext smc = (StreamMessageContext)context;
    InputStream ins = smc.getInputStream();
    ins.mark(1000);
    byte bytes[] = new
   byte[TestInputStream.SERVER_TO_CLIENT.length];
    ins.read(bytes);
    String s = new String(bytes);
    System.out.println("Got string: "+s);
    return true;
  }
}
```

CHAPTER 13

Artix IDL to Java Mapping

This chapter describes how Artix maps IDL to Java; that is, the mapping that arises by converting IDL to WSDL (using the IDL-to-WSDL compiler) and then WSDL to Java (using the WSDL-to-Java compiler).

In this chapter

This chapter discusses the following topics:

Introduction to IDL Mapping	page 260
IDL Basic Type Mapping	page 262
IDL Complex Type Mapping	page 264
IDL Module and Interface Mapping	page 277

Introduction to IDL Mapping

Overview

This chapter gives an overview of the Artix IDL-to-Java mapping. Mapping IDL to Java in Artix is performed as a two step process, as follows:

1. Map the IDL to WSDL using the Artix IDL compiler. For example, you could map a file, `SampleIDL.idl`, to a WSDL contract, `SampleIDL.wsdl`, using the following command:

   ```
   idl -wsdl SampleIDL.idl
   ```

2. Map the generated WSDL contract to Java using the WSDL-to-Java compiler. For example, you could generate Java stub code from the `SampleIDL.wsdl` file using the following command:

   ```
   wsdltojava SampleIDL.wsdl
   ```

For a detailed discussion of these command-line utilities, see the *Artix Command Line Reference Guide*.

Alternative Java mappings

If you are already familiar with CORBA technology, you will know that there is an existing standard for mapping IDL to Java directly, which is defined by the Object Management Group (OMG). Hence, two alternatives exist for mapping IDL to Java, as follows:

- Artix IDL-to-Java mapping—this is a two stage mapping, consisting of IDL-to-WSDL and WSDL-to-Java. It is an IONA-proprietary mapping.
- CORBA IDL-to-Java mapping—as specified in the OMG Java Language Mapping document (http://www.omg.org). This mapping is used, for example, by the IONA's Orbix.

These alternative approaches are illustrated in Figure 11.

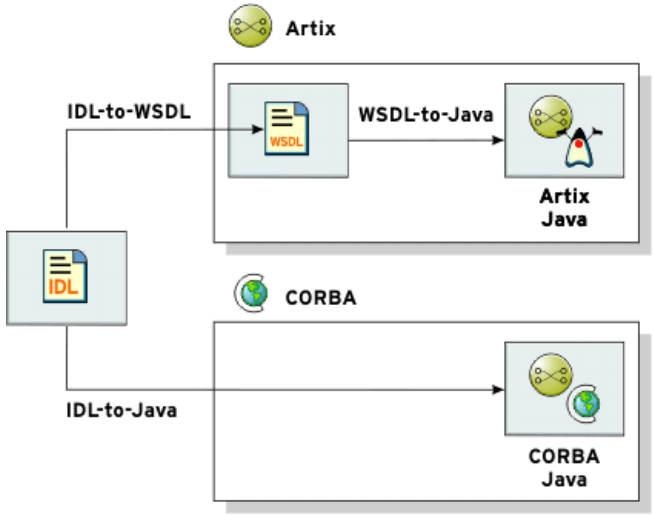

Figure 11: *Artix and CORBA Alternatives for IDL to Java Mapping*

The advantage of using the Artix IDL-to-Java mapping in an application is that it removes the CORBA dependency from your source code. For example, a server that implements an IDL interface using the Artix IDL-to-Java mapping can also interoperate with other Web service protocols, such as SOAP over HTTP.

Unsupported IDL types

The following IDL types are not supported by the Artix Java mapping:

- `long double`
- Value types
- Boxed values
- Local interfaces
- Abstract interfaces
- forward-declared interfaces

IDL Basic Type Mapping

Overview

Table 9 shows how IDL basic types are mapped to WSDL and then to Java.

Table 9: *Artix Mapping of IDL Basic Types to Java*

IDL Type	WSDL Schema Type	Java Type
any	xsd:anyType	com.iona.webservices.reflect.types.AnyType
boolean	xsd:boolean	boolean
char	xsd:byte	byte
string	xsd:string	java.lang.String
wchar	xsd:string	java.lang.String
wstring	xsd:string	java.lang.String
short	xsd:short	short
long	xsd:int	int
long long	xsd:long	long
unsigned short	xsd:unsignedShort	int
unsigned long	xsd:unsignedInt	long
unsigned long long	xsd:unsignedLong	java.math.BigInteger
float	xsd:float	float
double	xsd:double	double
octet	xsd:unsignedByte	short
fixed	xsd:decimal	java.math.BigDecimal

Mapping for string

The IDL-to-WSDL mapping for strings is ambiguous, because the `string`, `wchar`, and `wstring` IDL types all map to the same type, `xsd:string`. This ambiguity can be resolved, however, because the generated WSDL records the original IDL type in the CORBA binding description (that is, within the scope of the `<wsdl:binding> </wsdl:binding>` tags). Hence, whenever an `xsd:string` is sent over a CORBA binding, it is automatically converted back to the original IDL type (`string`, `wchar`, or `wstring`).

IDL Complex Type Mapping

Overview

This section describes how the following IDL data types are mapped to WSDL and then to Java:

- enum type
- struct type
- union type
- sequence types
- array types
- exception types
- typedef of a simple type
- typedef of a complex type

enum type

Consider the following definition of an IDL enum type, `SampleTypes::Shape`:

```
// IDL
module SampleTypes {
    enum Shape { Square, Circle, Triangle };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Shape` enum to a WSDL restricted simple type, `SampleTypes.Shape`, as follows:

```
<xsd:simpleType name="SampleTypes.Shape">
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Square"/>
       <xsd:enumeration value="Circle"/>
       <xsd:enumeration value="Triangle"/>
   </xsd:restriction>
</xsd:simpleType>
```

The WSDL-to-Java compiler maps the `SampleTypes.Shape` type to a Java class, `SampleTypesShape`, as shown in Example 178.

Example 178: *SampleTypesShape*

```
// Java
public class SampleTypeShape
{
  ...

  private final String _val;

  public static final String _Square = "Square";
  public static final SampleTypeShape Square = new SampleTypeShape(_Square);

  public static final String _Circle = "Circle";
  public static final SampleTypeShape Circle = new SampleTypeShape(_Circle);

  public static final String _Triangle = "Triangle";
  public static final SampleTypeShape Triangle = new SampleTypeShape(_Triangle);

  protected SampleTypeShape(String value)
  {
    _val = value;
  }

  public String getValue()
  {
    return _val;
  };

  public static SampleTypeShape fromValue(String value)
  {
        if (value.equals(_Square)) {
            return Square;
        }
        if (value.equals(_Circle)) {
            return Circle;
        }
        if (value.equals(_Triangle)) {
            return Triangle;
        }
        throw new IllegalArgumentException("Invalid enumeration value: "+value);
    };
```

Example 178:*SampleTypesShape*

```
    public static SampleTypeShape fromString(String value) {
        if (value.equals("Square")) {
            return Square;
        }
        if (value.equals("Circle")) {
            return Circle;
        }
        if (value.equals("Triangle")) {
            return Triangle;
        }
        throw new IllegalArgumentException("Invalid enumeration value: "+value);
    };

    public String toString() {
        return ""+_val;
    }

}
```

The value of the enumeration type can be accessed using the `getValue()` member function.

Programming with the Enumeration Type

For details of how to use the enumeration type, see "Enumerations" on page 98.

---

union type

Consider the following definition of an IDL union type, `SampleTypes::Poly`:

```
// IDL
module SampleTypes {
    union Poly switch(short) {
        case 1: short theShort;
        case 2: string theString;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::Poly` union to an XML schema choice complex type, `SampleTypes.Poly`, as follows:

```
<xsd:complexType name="SampleTypes.Poly">
    <xsd:choice>
        <xsd:element name="theShort" type="xsd:short"/>
        <xsd:element name="theString" type="xsd:string"/>
    </xsd:choice>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.Poly` type to a C++ class, `SampleTypesPoly`, as shown in Example 179.

Example 179: *SampleTypesPoly*

```
// Java
public class SampleTypesPoly {

...

    private String __discriminator;

    private short theShort;
    private String theString;

    public short getTheShort() {
        return theShort;
    }

    public void setTheShort(short _v) {
        this.theShort = _v;
        __discriminator = "theShort";
    }

    public boolean isSetTheShort() {
        if(__discriminator != null &&
           __discriminator.equals("theShort")) {
            return true;
        }

        return false;
    }
```

Example 179:*SampleTypesPoly*

```
    public String getTheString() {
        return theString;
    }

    public void setTheString(String _v) {
        this.theString = _v;
        __discriminator = "theString";
    }

    public boolean isSetTheString() {
        if(__discriminator != null &&
           __discriminator.equals("theString")) {
            return true;
        }

        return false;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("theShort: "+theShort+"\n");
        if (theString != null) {
            buffer.append("theString: "+theString+"\n");
        }
        return buffer.toString();
    }
}
```

The value of the union can be modified and accessed using the get*UnionMember*() and set*UnionMember*() pairs of functions.

Programming with the Union Type

For details of how to use the union type, see "Choice Complex Types" on page 64.

struct type

Consider the following definition of an IDL struct type, `SampleTypes::SampleStruct`:

```
// IDL
module SampleTypes {
    struct SampleStruct {
        string theString;
        long theLong;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStruct` struct to an XML schema sequence complex type, `SampleTypes.SampleStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SampleStruct">
    <xsd:sequence>
        <xsd:element name="theString" type="xsd:string"/>
        <xsd:element name="theLong" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.SampleStruct` type to a Java class, `SampleTypesSampleStruct`, as shown in Example 180.

Example 180: *SampleTypesSampleStruct*

```
//Java
public class SampleTypesSampleStruct {

...
    private String theString;
    private int theLong;

    public String getTheString() {
        return theString;
    }

    public void setTheString(String val) {
        this.theString = val;
    }
```

Example 180:*SampleTypesSampleStruct*

```
    public int getTheLong() {
        return theLong;
    }

    public void setTheLong(int val) {
        this.theLong = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (theString != null) {
            buffer.append("theString: "+theString+"\n");
        }
        buffer.append("theLong: "+theLong+"\n");
        return buffer.toString();
    }
}
```

The members of the struct can be accessed and modified using the get*StructMember*() and set*StructMember*() pairs of functions.

Programming with the Struct Type

For details of how to use the struct type in C++, see "Sequence and All Complex Types" on page 57.

---

sequence types

Consider the following definition of an IDL sequence type, `SampleTypes::SeqOfStruct`:

```
// IDL
module SampleTypes {
    typedef sequence< SampleStruct > SeqOfStruct;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SeqOfStruct` sequence to a WSDL sequence type with occurrence constraints, `SampleTypes.SeqOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.SeqOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.SeqOfStruct` type to a Java class, `SampleTypesSeqOfStruct`, as shown in Example 181.

Example 181: *SampleTypesSeqOfStruct*

```
// Java
public class SampleTypesSeqOfStruct {

    private SampleTypesSampleStruct[] item;

    public SampleTypesSampleStruct[] getItem() {
        return item;
    }

    public void setItem(SampleTypesSampleStruct[] val) {
        this.item = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (item != null) {
            buffer.append("item: "+Arrays.asList(item).toString()+"\n");
        }
        return buffer.toString();
    }
}
```

Programming with Sequence Types

For details of how to use sequence types, see "Sequence and All Complex Types" on page 57.

array types

Consider the following definition of an IDL union type, `SampleTypes::ArrOfStruct`:

```
// IDL
module SampleTypes {
     typedef SampleStruct ArrOfStruct[10];
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::ArrOfStruct` array to a WSDL sequence type with occurrence constraints, `SampleTypes.ArrOfStruct`, as follows:

```
<xsd:complexType name="SampleTypes.ArrOfStruct">
    <xsd:sequence>
        <xsd:element name="item"
            type="xsd1:SampleTypes.SampleStruct"
            minOccurs="10" maxOccurs="10"/>
    </xsd:sequence>
</xsd:complexType>
```

The WSDL-to-Java compiler maps the `SampleTypes.ArrOfStruct` type to a Java class, `SampleTypesArrOfStruct`, as shown in Example 182.

Example 182: *SampleTypesArrOfStruct*

```
//Java
public class SampleTypesArrOfStruct {

    private SampleTypesSampleStruct[] item;

    public SampleTypesSampleStruct[] getItem() {
        return item;
    }

    public void setItem(SampleTypesSampleStruct[] val) {
        this.item = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (item != null) {
            buffer.append("item: "+Arrays.asList(item).toString()+"\n");
        }
        return buffer.toString();
    }
}
```

Programming with Array Types

For details of how to use array types, see "Sequence and All Complex Types" on page 57.

---

exception types

Consider the following definition of an IDL exception type, `SampleTypes::GenericException`:

```
// IDL
module SampleTypes {
    exception GenericExc {
        string reason;
    };
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::GenericExc` exception to a WSDL sequence type, `SampleTypes.GenericExc`, and to a WSDL fault message, `_exception.SampleTypes.GenericExc`, as follows:

```
<xsd:complexType name="SampleTypes.GenericExc">
    <xsd:sequence>
        <xsd:element name="reason" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
...
<xsd:element name="SampleTypes.GenericExc"
    type="xsd1:SampleTypes.GenericExc"/>
...
<message name="_exception.SampleTypes.GenericExc">
    <part name="exception"
   element="xsd1:SampleTypes.GenericExc"/>
</message>
```

The WSDL-to-Java compiler maps the `SampleTypes.GenericExc` type to the Java class, `SampleTypesGenericExc`, as shown in Example 183.

Example 183: *SampleTypesGenericExc*

```
public class SampleTypesGenericExc {

    private String reason;

    public String getReason() {
        return reason;
    }

    public void setReason(String val) {
        this.reason = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (reason != null) {
            buffer.append("reason: "+reason+"\n");
        }
        return buffer.toString();
    }
}
```

In addition, the WSDL-to-Java compiler creates a class to map the message, `_exception.SampleTypes.GenericExc`, to a Java exception as shown in Example 184.

Example 184: *Java Exception*

```
public class SampleTypesGenericExcException extends Exception {
    private String reason;

    public SampleTypesGenericExcException(String reason) {
        super();
        this.reason = reason;
    }

    public SampleTypesGenericExcException() {
        super();
    }

    public String getReason() {
        return reason;
    }

    public void setReason(String val) {
        this.reason = val;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer(super.toString());
        if (reason != null) {
            buffer.append("reason: "+reason+"\n");
        }
        return buffer.toString();
    }
}
```

Programming with Exceptions in Artix

For an example of how to initialize, throw and catch a WSDL fault exception, see "Creating User-Defined Exceptions" on page 117.

typedef of a simple type

Consider the following IDL typedef that defines an alias of a `float`, `SampleTypes::FloatAlias`:

```
// IDL
module SampleTypes {
    typedef float FloatAlias;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::FloatAlias` typedef directory to the type, `xsd:float`. The WSDL-to-Java compiler then maps the `xsd:float` type directly to the `float` type.

typedef of a complex type

Consider the following IDL typedef that defines an alias of a `struct`, `SampleTypes::SampleStructAlias`:

```
// IDL
module SampleTypes {
    typedef SampleStruct SampleStructAlias;
    ...
};
```

The IDL-to-WSDL compiler maps the `SampleTypes::SampleStructAlias` typedef directly to the plain, unaliased `SampleTypes.SampleStruct` type.

The WSDL-to-Java compiler then maps the `SampleTypes.SampleStruct` WSDL type directly to the `SampleTypesSampleStruct` Java type. The Java mapping uses the original, unaliased type.

Note: The typedef of an IDL sequence or an IDL array is treated as a special case, with a specific class being generated to represent the sequence or array type.

IDL Module and Interface Mapping

Overview

This section describes the Artix Java mapping for the following IDL constructs:

- Module mapping
- Interface mapping
- Operation mapping
- Attribute mapping

Module mapping

An IDL identifier appearing within the scope of an IDL module, *ModuleName*::*Identifier*, maps to a Java identifier of the form *ModuleNameIdentifier*. That is, the IDL scoping operator, ::, is dropped in Java.

Although IDL modules do *not* map to packages under the Artix Java mapping, it is possible nevertheless to put generated Java code into a package using the -p switch to the WSDL-to-Java compiler (see "Generating Stub and Skeleton Code" on page 10). For example, if you pass a namespace, TEST, to the WSDL-to-Java -p switch, the *ModuleName*::*Identifier* IDL identifier would map to TEST.*ModuleNameIdentifier*.

Interface mapping

An IDL interface, *InterfaceName*, maps to a Java class of the same name, *InterfaceName*. If the interface is defined in the scope of a module, that is *ModuleName*::*InterfaceName*, the interface maps to the *ModuleNameInterfaceName* Java class.

If an IDL data type, *TypeName*, is defined within the scope of an IDL interface, that is *ModuleName*::*InterfaceName*::*TypeName*, the type maps to the *ModuleNameInterfaceNameTypeName* Java class.

Operation mapping

Example 185 shows two IDL operations defined within the `SampleTypes::Foo` interface. The first operation is a regular IDL operation, `test_op()`, and the second operation is a oneway operation, `test_oneway()`.

Example 185: *Example IDL Operations*

```
//IDL
module SampleTypes {

    interface Foo {
        string test_op(
            in long inLong,
            inout long inoutLong,
            out long   outLong
        );

        oneway void test_oneway(in string in_str);
    };
};
```

The operations from the preceding IDL, Example 185 on page 278, map to C++ as shown in Example 186.

Example 186: *Mapping IDL Operations to Java*

```
//Java
public class FooImpl {
1   public String test_op(
                        int inLong,
                    javax.xml.rpc.holders.IntHolder inoutLong,
                    javax.xml.rpc.holders.IntHolder outLong) {
      ...
    }
2 public void test_oneway(String in_str) {
      ...
    }
}
```

The preceding C++ operation signatures can be explained as follows:

1. The Java mapping of an IDL operation retains a similar signature to its IDL definition.

   The order of parameters in the Java method, `test_op()`, is determined as follows:

   - First, the `in` parameters appear in the same order as in IDL.
   - Next, the `inout` parameters appear in the same order as in IDL.
   - Finally, the `out` parameters appear in the same order as in IDL.

2. The Java mapping of an IDL oneway operation is straightforward, because a oneway operation can have only `in` parameters and a `void` return type.

Attribute mapping

Example 187 shows two IDL attributes defined within the `SampleTypes::Foo` interface. The first attribute is readable and writable, `str_attr`, and the second attribute is readonly, `bool_attr`.

Example 187: *Example IDL Attributes*

```
// IDL
module SampleTypes {
    ...
    interface Foo {
        ...
        attribute string               str_attr;
        readonly attribute boolean     bool_attr;
    };
};
```

The attributes from the preceding IDL, Example 187 on page 279, map to Java as shown in Example 188.

Example 188: *Mapping IDL Attributes to Java*

```
// Java
public class FooImpl {
```

Example 188:*Mapping IDL Attributes to Java*

```
    public String _get_str_attr() {
        // User code goes in here.
        return "";
    }

    public void _set_str_attr(String _arg) {
        // User code goes in here.
    }
public boolean _get_bool_attr() {
        // User code goes in here.
        return false;
    }
}
```

The preceding C++ attribute signatures can be explained as follows:

1. A normal IDL attribute, *`AttributeName`*, maps to a pair of accessor and modifier functions in Java, `_get_`*`AttributeName`*`()`, `_set_`*`AttributeName`*`()`.
2. An IDL readonly attribute, *`AttributeName`*, maps to a single accessor function in Java, `_get_`*`AttributeName`*`()`.

Glossary

A

anyType

anyType is the root type for all XMLSchema types. All of the primitive types are derivatives of this type, as are all user defined complex types.

Artix bus

The Artix bus reads the protocol details from the physical section of the Artix contract, loads the appropriate payload and transport plug-ins, and handles the mapping of the data onto and off the wire.

Artix message context

An Artix message context is a special message context that is used by Artix to store and transmit transport details and message header information. They contain two context containers. One for storing data about requests and one for storing data about replys. For more details see "Working with Artix Message Contexts" on page 211.

Artix reference

An Artix reference is a Java object that fully describes a running Artix service. References can be passed between Artix endpoints as operation parameters and are used extensively by the Artix locator. For more details see "Artix References" on page 145.

B

Binding

A binding maps an operation's messages to a payload format. Bindings are defined using the WSDL `<binding>` element. See also Payload format.

Bus

See Artix bus.

C

Choice complex type

A choice complex type is an XMLSchema construct defined by using a `<choice>` element to constrain the possible elements in a complex type. When using a choice complex type only one of the elements defined in the complex type can be valid at a time. For more details see "Choice Complex Types" on page 64.

Classloader firewall

The classloader firewall provides a user configurable way to block the Artix Java runtime from classes on a system's classpath. For more details see "Class Loading" on page 41.

Contract

An Artix contract is a WSDL file that defines the interface and all connection information for that interface.

A contract contains two components: *logical* and *physical*. The logical component defines things that are independent of the underlying transport and wire format such as abstract definitions of the data used and the interface.

The physical component defines the wire format, middleware transport, and service groupings, as well as the mapping between the operations defined in the interface and the wire formats, and the buffer layout for fixed formats and extensors.

D

Discriminator

A discriminator is a data element created to support the mapping of a choice complex type to a Java object. The discriminator element identifies the valid element in a choice complex type. See also Choice complex type.

Dynamic proxy

A dynamic proxy is a Java construct introduced in version 1.3 by Sun Micosystems. As specified by the JAX-RPC specification, Artix uses a dynamic proxy to connect to remote services. For more information, go to http://java.sun.com/reference/docs/index.html.

E

Embedded deployment

An embedded deployment is a deployment mode in which an application creates an endpoint, either by invoking Artix APIs directly, or by compiling and linking Artix-generated stubs and skeletons to connect client and server to the service bus.

Endpoint

The runtime incarnation of a service defined in an Artix contract. When using the Artix Java APIs, an endpoint is activated when you register a servant with the Artix bus. See also Service.

F

Facet

A facet is a rule used in the derivation of user defined simple types. Common facets include `length`, `pattern`, `totalDigits`, and `fractionDigits`. For more details see "Defining Your Own Simple Types" on page 53.

Factory pattern

The factory pattern is a usage pattern where one service creates and manages instances of another service. Typically, the factory service returns references to the services it creates. For more details see "Using References in a Factory Pattern" on page 154.

Fault message

A fault message is the WSDL construct used to define error messages, or exceptions, passed between a service and its clients. They are defined using a `<fault>` element in a WSDL contract. For more details see "Creating User-Defined Exceptions" on page 117.

H

Handler

`Hanlder` is the Java interface that a developer must implement to create a message handler. It has has methods for processing both request and response messages. Artix provides a `GenericHandler` class to provide a template for implemeting message handlers. See also "Writing Message Handlers" on page 239.

I

Input message

An input message is the WSDL construct for defining the messages that are sent from a client to a service and are specified using an `<input>` element in a WSDL contract. When mapped into Java, the parts of the input message are mapped into a method's parameter list.

Interface

An interface defines the operations offered by a service. Interfaces are defined in an Artix contract using the WSDL `<portType>` element. When mapped to Java, an interface results in the generation of an object with methods for each of the operations defined in the interface. See also Operation.

J

Java API for XML-Based RPC(JAX-RPC)

JAX-RPC is the Java specification upon which Artix based it Java API and data type mappings. For more information go to http://java.sun.com/xml/jaxrpc/overview.html.

L

List type

A list type is a data type defined as consisting of a space separated list of primitive type elements. For example, "1 2 3 4 5" is a valid value for a list type. They are defined using a `<xsd:list>` element. For more details see "Lists" on page 95.

Logical contract

The logical contract defines components that are independent of the underlying transport and wire format. These include the type definitions and the interface definitions. WSDL elements found in the logical contract include: `<portType>`, `<operation>`, `<message>`, `<type>`, and `<import>`.

M

Message

In Artix, a message is any data passed between two endpoints. Messages are defined in an Artix contract using the WSDL `<message>` element and are used for the input, output, and fault messages that define an operation. After a message has been associated with an operation, it can be bound to any payload format supported by Artix. See also Fault message, Input message, and Output message.

Message-level handler

A message-level handler is a message handler that processes messages between the Artix binding to the Artix transport. See "Writing Message Handlers" on page 239.

Message context

A message context is a bus container used by applications to store metadata properties. These properties store information about the message being sent went an operation is invoked. Artix uses the message context to store headers and transport information. See also Artix message context and "Using Message Contexts" on page 197.

Message handler

A message handler is a Java class responsible for intercepting a message along the message chain and performing some processing on the raw message data. See also Handler and "Writing Message Handlers" on page 239.

O

Operation

An operation defines a specific interaction between a service and a client. It is defined in an Artix contract using the WSDL `<operation>` element. Its definition must include at least one input or output message. When mapped into Java, an operation generates a method on the object representing the interface in which it is defined.

Output message

An output message is the WSDL construct for defining the messages that are sent from a service to a client and are specified using an `<output>` element in a WSDL contract. When mapped into Java, the parts of the output message are mapped as described in the JAX-RPC specification.

P

Payload format

A payload format is how data is packaged to be sent on the wire. Examples of payload formats supported by Artix include SOAP, TibMsg, and fixed record length data. Data is bound to a payload format in an Artix contract using the WSDL `<binding>` element.

Physical contract

The physical contract defines the bindings and transport details used by the endpoints defined by an Artix contract. WSDL elements found in the physical contract include: `<binding>`, `<service>`, and `<port>`.

Plug-in

A plug-in is a module that Artix loads at runtime to provide a set of features. All of the bindings and transports supported by Artix are implemented as plug-ins. In addition, message handlers are implemented as plug-ins.

R

Reply

A reply is the message returned by a service to a client in response to a request from the client. See also Output message.

Request

A request is a message sent from a client to a service asking for the service to do work. See also Input message.

Request-level handler

A request-level handler is a message handler that processes messages between the Artix binding and the user's application code. See "Writing Message Handlers" on page 239.

Response

See Reply.

S

Servant

A servant is a Java object that wraps the implementation object generated from an interface. The servant wrapper enables the bus to associate the implementation object with the physical details specified in its contract's service definition and to manage the object.

Service

A service is the contract definition of an Artix endpoint. It combines the logical definition of an interface, the binding of the interface's operations to a payload format, and the transport details used to expose the interface. A service is defined using a WSDL `<port>` element.

Service proxy

A service proxy is a proxy created by an Artix client to connect to a remote service. See also Dynamic proxy.

Service template

A service template is a WSDL service definition that serves as the model for the colnes created for a transient reference. They must fully define all of the details, except the address, of the transport used by the transient servant. The address provided in the service template must be a wildcard value.

Standalone deployment

Standalone deployment is a deployment mode in which an Artix instance runs independently of the endpoints it is integrating.

Static servant

A static servant is a servant whose physical details are linked to a `<port>` definition in the contract associated with the application. For more details see "Static Servant Registration" on page 27.

Stub interface

Artix service proxies implement the `javax.xml.rpc.Stub` interface. The Stub interface provides access to a number of low-level properties used to connect the proxy to a remote service. These properties can be used to get the Artix bus from client applications, to register type factories, and set HTTP connection properties.

T

Transient servant

A transient servant is a servant whose physical details are cloned from a `<port>` definition in the contract associated with the application. For more details see "Transient Servant Registration" on page 28.

Transport

A transport is the network protocol, such as HTTP or IIOP, that is used by an endpoint. The transport details for an endpoint are defined inside of the WSDL `<port>` element defining the endpoint.

Type factory

A type factory is a Java class generated to support the use of XMLSchema anyTypes and SOAP headers in Java.

W

Web Service Definition Language(WSDL)

WSDL is an XML format for describing network services as a set of endpoints. Artix uses WSDL as the syntax for its contracts.

In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data binding formats. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- Types -- a container for data type definitions using some type system (such as XMLSchema).
- Message -- an abstract, typed definition of the data being communicated.
- Operation -- an abstract definition of an action supported by the service.
- Port Type -- an abstract set of operations supported by one or more endpoints.
- Binding -- a concrete protocol and data format specification for a particular port type.
- Port -- a single endpoint defined as a combination of a binding and a network address.
- Service -- a collection of related endpoints.

For more information go to http://www.w3.org/TR/wsdl.

WSDL <binding>

See Binding and Payload format.

WSDL <fault>

See Fault message.

WSDL <message>

See Message.

WSDL <operation>
See Operation.

WSDL <port>
See Service.

WSDL <portType>
See Interface.

WSDL <service>
A WSDL `<service>` element is a collection of WSDL `<port>` elements.

X

XMLSchema
XMLSchema is a language specification by the W3C that defines an XML meta-language for defining the contents and structure of XML documents. It is used as the native type system for Artix. For more information go to http://www.w3.org/XML/Schema.

Index

A

abstract interface type 261
AnyType
  getBoolean() 141
  getByte() 141
  getDecimal() 141
  getDouble() 141
  getFloat() 141
  getInt() 141
  getLong() 141
  getSchemaTypeName() 140
  getShort() 141
  getString() 141
  getType() 142
  getUByte() 141
  getUInt() 141
  getULong() 141
  getUShort() 141
  setBoolean() 138
  setByte() 138
  setDecimal() 139
  setDouble() 138
  setFloat() 138
  setInt() 138
  setLong() 138
  setShort() 138
  setString() 138
  setType() 139
  setUByte() 138
  setUInt() 139
  setULong() 139
  setUShort() 138
anyType 136
arrayType attribute 93
Artix bus 3
  initializing 16, 20
  starting 18
Artix locator
  overview 183
Artix services
  locator 187

B

binding name
  specifying to code generator 11
boxed value type 261
Bus
  createClient() 30
  createReference() 151
  getTypeFactoryMap() 129
  init() 16, 20
  registerTypeFactory() 129
  run() 18, 20
  shutdown() 21
bus
  registerHandlerFactory() 245, 252
BusPlugIn 234
BusPlugIn.busInit() 234
BusPlugIn.busShutdown() 235
BusPlugIn.getBus() 234
BusPlugInFactory 237
BusPlugInFactory().createBusPlugIn() 237

C

client
  developing 20
client proxy
  instantiating 20
client stub code 10
code generation 10
  from the command line 11
  impl flag 15
  server flag 16
  types flag 15
code generator
  command-line 11
  files generated 10
com.iona.jbus.Servant 17
com.iona.jbus package 13
com.iona.webservices.reflect.types.AnyType 137
com.iona.webservices.reflect.types.TypeFactory 12
    7, 137
complex choice type
  receiving 64
  transmitting 64

complex types
  attributes 68
  derivation by extension 104
  derivation by restriction 78
  deriving from simple 78
  description in XMLSchema 56
  mapping to Java 56
configuration
  -ORBname switch 191
ContextRegistry 201
context registry 201
contexts
  stub files, generating 220
  type factories for 221
contract type descriptions 53, 56
CORBA
  abstract interface 261
  any 262
  basic types 262
  boolean 262
  boxed value 261
  char 262
  enum type 264
  exception type 273
  fixed 262
  forward-declared interfaces 261
  local interface type 261
  sequence type 270
  string 262
  struct type 269
  typedef 276
  union type 266, 272
  value type 261
  wchar 262
  wstring 262
createClient() 30, 153, 195
createReference() 151, 152
createService() 21
creating a dynamic proxy 21
creating a Service instance 21

D

developing a server 15
dynamic proxies 20
dynamic proxy
  instantiating 20

E

EndpointNotExist fault 189
endpoints 185
  registering with the locator 191
enum type 264
exception handling
  CORBA mapping 274
exceptions
  associating to an operation 119
  describing in a contract 118
exception type 273

F

facets 53
fault message 5
forward-declared interfaces 261
fractionDigits facet 55
fromString() 99
fromValue() 99

G

generated getter method 58
generated setter method 58
generated types
  getter method 58
  setter method 58
GenericHandler 248, 255
GenericHandlerFactory 246, 253
getBoolean() 141
getByte() 141
getClass() 140
getClientMessageHandler() 253
getContextRegistry() 201
getCurrent() 203
getDecimal() 141
getDouble() 141
getFloat() 141
getInt() 141
getJavaType() 133
getJavaTypeForElement() 134
getLong() 141
getReplyContext() 214
getReplyContextAsString() 214
getRequestContext() 214
getRequestContextAsString() 214
getSchemaType() 132
getSchemaTypeName() 140
getServerMessageHandler() 253

getShort() 141
getString() 141
getSupportedNamespaces() 131
getType() 142
getTypeFactoryMap() 129
getTypeResourceLocation() 134
getUByte() 141
getUInt() 141
getULong() 141
getUShort() 141
getValue() 99

H

Handler 248, 255
  handleRequest() 248, 255
  handleResponse() 248, 255
handleRequest() 248, 255
handleResponse() 248, 255
HandlerFactory 246, 253
  getClientMessageHandler() 253
  getServerMessageHandler() 253
HandlerInfo 247, 254
  setHandlerClass() 247, 254
http plug-in 191

I

IDL
  enum type 264
  exception type 273
  oneway operations 279
  sequence type 270
  struct type 269
  typedef 276
  union type 266, 272
IDL attributes
  mapping to Java 279
IDL basic types 262
IDL interfaces
  mapping to Java 277
IDL modules
  mapping to Java 277
IDL operations
  mapping to C++ 278
  parameter order 279
  return value 279
IDL readonly attribute 280
IDL-to-C++ mapping
  Artix and CORBA 260
IDL types
  unsupported 261
idl utility 260
init() 16, 20
  -ORBname parameter 194
initializing the bus
  client side 20
  server side 16
input message 5
instantiating a client proxy 20
IonaMessageContext 203, 205
itemType 95
itemType attribute 97

J

java.io.* package 14
java.net.* package 14
java.rmi.Remote 6
java.rmi.RemoteException exception 7
Java Exception class 120
Java Holder class 7
javax.activation.DataHandler 114
javax.xml.namespace.QName package 13
javax.xml.rpc.* package 13
javax.xml.rpc.holders 107
javax.xml.rpc.holders.Holder interface 107
javax.xml.rpc.holders package 7
javax.xml.rpc.security.auth.password 38
javax.xml.rpc.security.auth.username 38
javax.xml.rpc.service.endpoint.address 39
javax.xml.rpc.ServiceFactory 20
javax.xml.rpc.Service interface 20
javax.xml.soap.Name 87
javax.xml.soap.Node 88
javax.xml.soap.SOAPElement 86
javax.xml.soap.Text 88

L

length facet 55
list types 95
load balancing
  with the locator 184
local interface type 261
locator
  binding and protocol 187
  embedded deployment 185
  EndpointNotExist fault 189
  load balancing 184, 186

LocatorService port type,Java mapping 189
lookupEndpointResponse type 189
lookupEndpoint type 189
reading a reference from 192
registering endpoints 191
standalone deployment 185
WSDL contract 187
locator, Artix 183
locator_endpoint plug-in 191
LocatorService port type 189
logical contract 2
lookupEndpointResponse type 189
lookupEndpoint type 189

M

mapping
IDL attributes 279
IDL interfaces 277
IDL modules 277
IDL operations 278
IDL to C++ 260
maxExclusive facet 55
maxInclusive facet 55
maxLength facet 55
MessageContext 203, 204
getProperty() 208
removeProperty() 209
setProperty() 206
message context 203
message part sharing 107
MIME multi-part related message 111
minExclusive facet 55
minInclusive facet 55
minLength facet 55
Multi-dimensional arrays 94

O

obtaining a ServiceFactory 21
occurrence constraints
overview of 81
oneway operations
in IDL 279
-ORBname, parameter to IT_Bus::init() 194
-ORBname command-line parameter 191
output message 5

P

parameters
in IDL-to-Java mapping 279
partially transmitted arrays
SOAP arrays
partially transmitted 94
pattern facet 55
PerInvocationServant 35
physical contract 2
plug-ins
http 191
locator_endpoint 191
soap 191
port
specifying to code generator 11
ports
and endpoints 185
portType 11
primitive types
Java 48
proxies
constructor for references 195

R

receiving choice types 64
ref:Reference type 189
references
constructor for client proxies 195
looking up in the locator 185
reading from the locator 192
ref:Reference type 189
schema 189
registerContext() for SOAP 221
registerHandlerFactory() 245, 252
registering a servant instance 18
registerServant() 18, 27
registerTransientServant() 29
registerTypeFactory() 129
removeReplyContext() 215
removeRequestContext() 216
reply context container 211
request context container 211
required java packages 13
run() 18, 20

S

schema
for references 189
sequence complex types 57
sequence type 270

SerializedServant 35
server
  developing 15
  implementation class 15
  main() function 16
server skeleton code 10
Service.getPort() 21
ServiceFactory.newInstance() 21
service name
  specifying to code generator 11
setBoolean() 138
setByte() 138
setDecimal() 139
setDouble() 138
setFloat() 138
setHandlerClass() 247, 254
setInt() 138
setLong() 138
setReplyContext() 212
setReplyContextAsString() 212
setRequestContext() 212
setRequestContextAsString() 212
setShort() 138
setString() 138
setType() 139
setUByte() 138
setUInt() 139
setULong() 139
setUShort() 138
shutdown() 21
shutting down the bus 21
simple types
  XMLSchema 48
SingleInstanceServant 34
skeleton code
  generating with wsdltojava 11
SOAP arrays
  sparse 94
  syntax 92
SOAP bindings 187
SOAPElement.getChildElements() 88
SOAPElement.getElementName() 87
SOAP-ENC:Array type 92
soap plug-in 191
SOAP with attachments 111
sparse arrays 94
static servant 27
StreamMessageContext 255
struct type 269
Stub._getProperty() 37
Stub._setProperty() 37
Stub interface 37

T

thread_pool:high_water_mark 33
thread_pool:initial_threads 33
thread_pool:low_water_mark 33
toString() 58, 99, 120
totalDigits facet 55
transient servant 28
transmitting choice types 64
typedef 276
type derivation
  by extension 78, 104
  by restriction 78
type factories 126
  and contexts 221
  generating 126
  instantiating 128
  registering 129
TypeFactory
  getJavaType() 133
  getJavaTypeForElement() 134
  getSchemaType() 132
  getSupportedNamespaces() 131
  getTypeResourceLocation() 134

U

union type 266, 272
unsupported IDL types 261

V

value type 261

W

whiteSpace facet 55
wsdl:arrayType 92
wsdl:arrayType attribute 93
WSDL <fault> element 7, 119
  message attribute 119
WSDL <input> element 7
WSDL <message> element 4, 7, 118
  name attribute 120
WSDL <operation> element 4, 7
  name attribute 7
  parameterOrder attribute 7

WSDL <output> element 7
WSDL <part> element 4
WSDL <port> element 6
  name attribute 6
WSDL <portType> element 4, 6
WSDL <types> element 4, 53, 56, 136
WSDL faults 274
wsdltojava 11, 15
  command-line switches 11
  -datahandlers 114
  files generated 10
  XML schemas, generating from 220
wsdltojava utility 260

X

XMLSchema <all> element 57
XMLSchema <attribute> element 50, 68
  default attribute 50, 69
  fixed attribute 50, 69
  name attribute 68
  type attribute 68
  use attribute 50, 68
XMLSchema <choice> element 64
XMLSchema <complexContent> element 104
XMLSchema <complexType> element 56
  name attribute 57, 73
XMLSchema <element> element 50
  maxOccurs attribute 50, 59, 81, 93
  minOccurs attribute 50, 81
  nillable attribute 50
  type attribute 72
XMLSchema <extension> element 78, 104
  base attribute 104
XMLSchema <restriction> element 53
  base attribute 53
XMLSchema <sequence> element 57
XMLSchema <simpleContent> element 78
XMLSchema <simpleType> element 53
  name attribute 53, 99
XMLSchema facets 53
xsd:anyType 136
  and context types 219
xsd:list 95