Artix™

Developing Artix Applications in Java

Version 3.0, June 2005

Making Software Work Together™

Updated: 04-Nov-2005

Contents

Part II Advanced Artix Programming

List of Figures

List of Tables

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.

The remainder of the book discusses advanced programming features of the Artix Java APIs such as handlers, persistence, and transactions. The chapters assume familiarity with the basic material covered in chapters 1 through 5. In addition, they assume a basic understanding of distributed system development.

Finding Your Way Around the Library

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

If you are new to Artix

You may be interested in reading:

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- Getting Started with Artix describes basic Artix and WSDL concepts.

To design and develop Artix solutions

Read one or more of the following:

- Designing Artix Solutions provides detailed information about describing services in Artix contracts and using Artix services to solve problems.
- Developing Artix Applications in C++ discusses the technical aspects of programming applications using the C++ API.

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

To configure and manage your Artix solution

Read one or more of the following:

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

Reference material

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

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

Have you got the latest version?

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

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

Searching the Artix Library

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

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

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

http://www.iona.com/support/docs/artix/3.0/index.xml

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

Online Help

Artix Designer includes comprehensive online help, providing:

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

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

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

Additional Resources

The IONA Knowledge Base contains helpful articles written by IONA experts about Artix and other products.

The IONA Update Center contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to IONA Online Support.

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

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

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

Keying Conventions

This book uses the following keying conventions:

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

Part I

Fundamentals of Artix Programming

In this part

This part contains the following chapters:

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

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 using a `portType` element. 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.

Operations

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.

portTypes

For each `portType` 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 `portType` element. The interface's name will be identical to the `portType` elements's name unless the `portType` element's name ends in `PortType`. In this case, the `PortType` will be stripped off the interface's name.

The generated interface will contain each of the operations of the `portType` to which the `portType` 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 in-out 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 59.

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 two 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="finalScore">
    <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 finalScore(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

Overview

The Artix development tools include a utility to generate server skeleton and client stub code from an Artix contract. In addition, Artix maps WSDL types to Java classes using the mapping described in the JAX-RPC specification.

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.
- *portTypeName*`TypeFactory.java` defines the type factories used by Artix to support the complex types used by the service.
- *portTypeName*`Demo.java` is a simple main class for a client.

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 59.

Generating code from the command line

You generate code at the command line using the command:

```
wsdltojava [-e service:port][-b binding][-i portType]
           [-d output_dir][-p [namespace=]package]
           [-impl][-server][-client][-plugin][-servlet]
           [-types][-call][-interface][-sample][-all][-ant]
           [-datahandlers][-merge][-deployable]
           [-nexclude namespace[=package]]
           [-ninclude namespace[=package]][-L file][-ser]
           [-q][-h][-V] 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`*`:`*`port`*	Specifies the name of the service, and optionally the port, for which the tool will generate code. The default is to use the first service listed in the contract. Specifying multiple services results in the generation of code for all the named service/port combinations. If no port is given, all ports defined in a service will be activated.
`-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`.
`-plugin`	Generate a bus plug-in with the appropriate servant registration code for the generated service implementation. When using this flag, the server mainline does not include code for registering the servant with the bus.

`-servlet`	Generates a bus plug-in with the additional information needed to deploy it as a servlet.
`-types`	Generates the code to implement the complex types defined by the contract.
`-call`	Generates a sample client the uses the `Call` interface to invoke on the remote service. For more information see "Using the Call Interface for Dynamic Invocations" on page 461.
`-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 information see "Using SOAP with Attachments" on page 151 and Designing Artix Solutions.
`-merge`	Merge any user changes into the generated code.
`-deployable`	Generate a deployment descriptor to deploy the generated plug-in into an Artix container. For more information see Deploying and Managing 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.
`-L` *`file`*	Specifies the location of your Artix license file. The default behavior is to check `IT_PRODUCT_DIR\etc\license.txt`.

`-ser`	Specifies that the generated classes for data types defined in the contract will be serializable (i.e. they will implement `java.io.Serializable`).
`-q`	Specifies that the tool runs in quiet mode. No output will be shown on the console. This includes error messages.
`-h`	Specifies that the tool will display a usage message.
`-v`	Specifies that the tool runs in verbose mode.

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 21 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(new HelloWorldImpl(),
                            "./HelloWorld.wsdl", 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\3.0\it_bus-api.jar
- *InstallDir*\lib\artix\ws_common\3.0\it_wsdl.jar
- *InstallDir*\lib\artix\ws_common\3.0\it_ws_reflect.jar
- *InstallDir*\lib\artix\ws_common\3.0\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 *InstallDir*/lib/sun/activation/1.0.1/activation.jar on your classpath.

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

CHAPTER 3

Things to Consider when Developing Artix Applications

Several areas must be considered when programming complex Artix applications.

In this chapter

This chapter discusses the following topics:

Bootstrapping Service

Overview

When it comes to deploying applications in a real system, it is typically inconvenient to hardcode the location of a contract in the application. It is more practical to specify the location of basic resources, such as a contract, at runtime—for example, by specifying the contract URL in configuration or on the command line.

The Artix *bootstrapping service* simplifies the process of obtaining the following kinds of basic resource: contracts and Artix references. The process is divided into two independent steps:

1. *Provide the basic resource*—you can provide a contract or an Artix reference in several different ways: by configuration, by specifying the location on the command line, and so on.
2. *Retrieve the basic resource*—Java functions are provided to retrieve WSDL `services` and Artix references, based on the qualified name (QName) of the resource.

In this section

This section discusses the following topics:

Finding Initial References

Overview

An Artix reference encapsulates the data required for creating a service proxy to connect to an Artix endpoint (essentially, this data is identical to the data contained in a WSDL `service` element). Once an applicaiton has a reference, it creates a servrice proxy by passing the reference to a proxy constructor.

The Artix bootstrapping service provides an API function, `Bus.resolveInitialReference()`, for finding initial references based on the QName of a WSDL `service`.

Example of finding an initial reference

Given that the bus has already loaded and parsed either an Artix reference (or a contract) containing a service called `SOAPService` in the namespace, `http://www.iona.com/hello_world_soap_http`, you can initialize a service proxy, `proxy`, as shown in Example 12.

Example 12: *Finding an Initial Reference Using the Bootstrapping Service*

```
QName name = new
 QName("http://www.iona.com/hello_world_soap_http",
       "SOAPService");

Reference ref;

// Find the initial reference using the bootstrap service
ref = bus.resolveInitialReference(name);

// Create a proxy and use it
GreeterClient proxy = (GreeterClient)bus.CreateClient(
                        ref,
                        GreeterClient.class);

proxy.sayHi();
```

Options for bootstrapping references

The bootstrapping service finds initial references from the following sources, in order of priority:

1. *Collocated service*—if the client code that calls `resolveInitialReference()` is collocated with (that is, in the same process as) the required service, the `resolveInitialReference()` function returns a reference to the collocated service. This assumes that the client and server code are using the same bus instance.
2. *References specified on the command line*—you can provide an initial reference by specifying on the command line the location of a file containing an Artix reference. For example:

   ```
   java bsServer -BUSinitial_reference ../../etc/hello_ref.xml
   ```

3. *References specified in the configuration file*—you can provide an initial reference from the configuration file, either by specifying the location of an Artix reference file or by specifying the literal value of an Artix reference.

   For more details, see Deploying and Managing Artix Solutions.
4. *Service in a contract*—the `service` element in a contract contains essentially the same data as an Artix reference. Hence, if a reference is not specified using one of the other methods, Artix searches any loaded contracts to find the specified service.

   The sources of contracts are the same as on the server side. The mechanism for bootstrapping references is, thus, effectively an extension of the mechanism for bootstrapping contracts—see "Options for bootstrapping WSDL" on page 34.

Finding Contracts

Overview

An Artix contract is required to:

- register a servant with the bus.
- create a service proxy using the JAX-RPC `Service` interface.

Registering a servant with the bus associates an implementation (represented by a servant object) with a particular WSDL `service`. The `Service` interface uses the information in a WSDL `service` to identify the operations exposed by the service and to open the proper network connection. The WSDL `service` must, therefore, be available from one of the contracts provided by the bootstrapping service.

The Artix bootstrapping service provides an API function, `Bus.getServiceWSDL()`, for retrieving the contract for a particular WSDL `service`. `getServiceWSDL()` takes the `QName` of the service and returns a string representing the location of the corresponding contract.

Example of finding a WSDL contract

Given that the bus has already loaded and parsed a contract containing the service, `SOAPService`, in the namespace, `http://www.iona.com/hello_world_soap_http`, you can find the WSDL `service` element as shown in Example 13.

Example 13: *Finding a WSDL Contract Using the Bootstrapping Service*

```
QName name = new
 QName("http://www.iona.com/hello_world_soap_http",
       "SOAPService");

// Find the WSDL contract using the bootstrap service
String wsdl = bus.getServiceWSDL(name);
```

Options for bootstrapping WSDL

The bootstrapping service finds contracts from the following sources, in order of priority:

1. *Contract specified on the command line*—you can provide a contract by specifying the location of the contract file on the command line. For example:

   ```
   java bsServer -BUSinitial_contract ../../etc/hello.wsdl
   ```

2. *Contract specified in the configuration file*—you can provide a contract from the configuration file. For example:

   ```
   # Artix Configuration File
   bus:qname_alias:hello_service =
      "{http://www.iona.com/hello_world_soap_http}SOAPService";
   bus:initial_contract:url:hello_service =
      "../../etc/hello.wsdl";
   ```

   This associates a nickname, `hello_service`, with the QName for the `SOAPService` service. The `bus:initial_contract:url:hello_service` variable then specifies the location of the WSDL contract containing this service.

   For more details, see Deploying and Managing Artix Solutions.

3. *Contract directory specified on the command line*—you can provide a contract by specifying a contract directory on the command line. When the bootstrapping service looks for a particular WSDL `service`, it searches all of the WSDL files in the specified directory. For example:

   ```
   java bsServer -BUSservice_contract_dir ../../etc/
   ```

   For more details, see Deploying and Managing Artix Solutions.

4. *Contract directory specified in the configuration file*—you can provide a contract by specifying a contract directory in the configuration file. For example:

   ```
   # Artix Configuration File
   bus:initial_contract_dir = [".", "../../etc"];
   ```

5. *Stub WSDL shared library*—the bootstrapping service can retrieve a contract that has been embedded in a shared library.

   Currently, this mechanism is *not* publicly supported. However, it is used internally by the following Artix services: Locator Service, Session Manager Service, Peer Manager, and Container Service.

References

For more details about how to register servants, see "Servant Registration" on page 36.

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

Overview

When a servant is registered as a *static* servant it is linked to a `port` element 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 14.

Example 14: *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 15 shows the code for registering a static servant.

Example 15: *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 Tunnel.

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 adhere 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 an HTTP service must be specified using *`host_name`*`:0`.

- 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 16.

Example 16: *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 17 shows the code for registering a transient servant.

Example 17: *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);
```

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 18.

Example 18: *Bus.createClient()*

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

Example

Example 19 shows the code for creating a service proxy using `createClient()`.

Example 19: *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 19 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` element defining the transport the proxy will use to contact the service. In this example, the transport details are defined in a `port` element 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 default 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 you will need to perform additional steps to get the bus.

Inside a service implementation object

If you are in a service's implementation object, you can use the code shown in Example 20.

Example 20: *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 21.

Example 21: *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 multi-threaded 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 44. The `SingleInstanceServant` provides no thread safety for the user code.

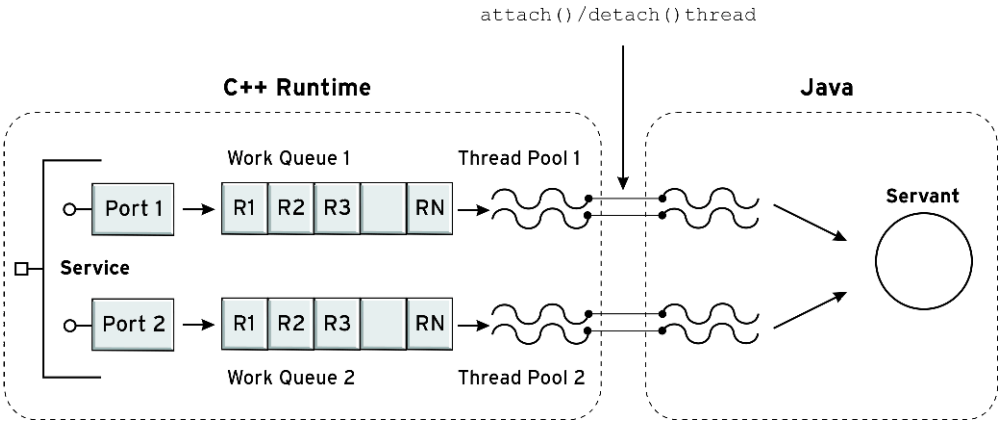

Figure 1: *SingleInstanceServant*

To instantiate 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 22 shows an example of instantiating a `SingleInstanceServant`.

Example 22: *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 45. Using a `SerializedServant` is equivalent to using a `SingleInstanceServant` whose target object is completely synchronized.

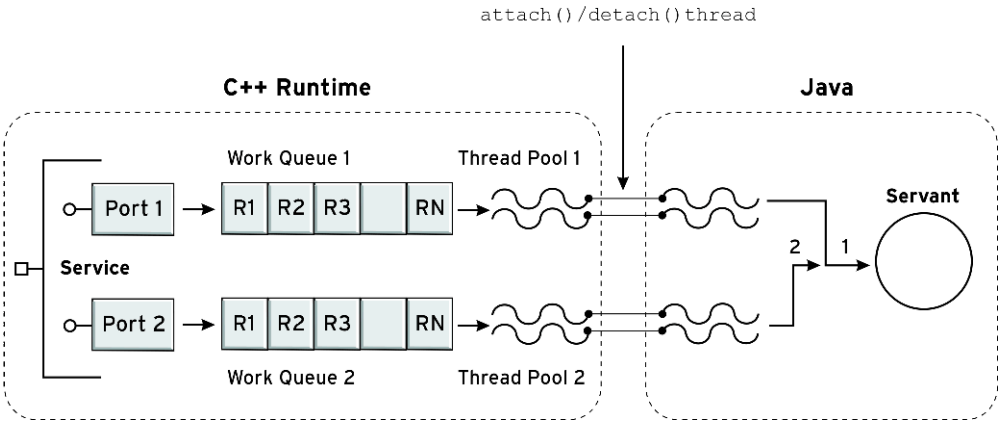

Figure 2: *SerializedServant*

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 22 shows an example of instantiating a `SerializedServant`.

Example 23: *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 46. This ensures thread safety, but does not allow the implementation object to have any statefull information.

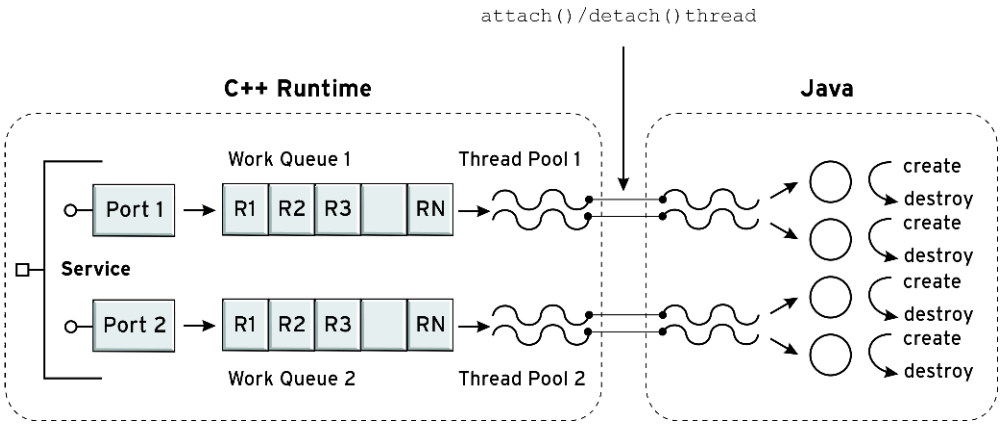

Figure 3: *PerInvocationServant*

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 24 shows the code for instantiating a `PerInvocationServant`.

Example 24: *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 25 shows code getting a `Stub` object from an Artix proxy.

Example 25: *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 25 on page 47.
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 26 on page 48 shows code for setting the username for a client.

Example 26: *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 25 on page 47.
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 27 on page 49 shows code for setting the password for a client.

Example 27: *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 25 on page 47.
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 27 on page 49 shows code for setting the endpoint address property for a client.

Example 28: *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);
```

Creating a Service Proxy Using UDDI

Overview

You can create a service proxy by dynamically locating existing web services' endpoints through a UDDI service. When an application does not have a pointer or reference to an instance of a running web service, Artix can take a service description then query a UDDI registry for an available service instance. The UDDI registry returns endpoint information that Artix uses to create a service proxy to invoke upon a specific instance of the service.

UDDI queries

Artix uses UDDI query strings that take the form of a URL. The syntax for a UDDI URL is shown in Example 29. The syntax adheres to the rules for URL syntax described in RFC2396 (Uniform Resource Identifiers (URI): Generic Syntax).

Example 29: *UDDI URL Syntax*

```
uddi:UDDIRegistryEndptURL?query
```

`UDDIRegistryEndptURL` specifies the HTTP URL of the UDDI registry that Artix is going to submit the query for a service endpoint. For example, you could deploy a local UDDI registry at the address `http://localhost:9000/uddi/inquiryapi`.

`query` is a string that Artix uses to look-up services in the UDDI registry. The query string specifies the UDDI attributes and their coresponding values to use in selecting an appropriate service from the registry. If more than one service in the registry match the query, Artix uses the first one found to create the service proxy. For example to return a widget ordering service, you could use the query string `tmodelname=widgetVendor`.

Note: Currently, only the `tmodelname` attribute is supported by Artix.

Example 30 shows a complete UDDI URL.

Example 30: *Artix UDDI URL*

```
uddi:http://localhost:9000/uddi/inquiryapi?tmodelname=widgets
```

Getting the service proxy

Using a UDDI registry to look up a service's endpoint information and using the returned endpoint information to create a service proxy is simple in Artix. The only change to your client application code is the path used to specify your contract location when creating the `Service` object or when calling `createClient()`.

In place of the location of an actual contract, you would use a UDDI URL to locate the service's contract. Artix will recognize the UDDI URL, query the UDDI registry, retrieve the service's endpoint information, and build the service proxy under the covers. Example 31 shows an example of creating a service proxy using UDDI.

Example 31: *Creating a Service Proxy with UDDI*

```
1  String query =
      "uddi:http://localhost:9090/uddi/inquiry?tmodelname=collie";

   URL wsdlURL;
   try
   {
     wsdlURL= new URL(query);
   } catch (java.net.MalformedURLException ex)
   {
     wsdlURL= new File(query).toURL();
   }

2  QName name = new QName("http://dogLova.com/borderCollies",
                          "SOAPAccess");

3  ServiceFactory factory = ServiceFactory.newInstance()

4  Service = factory.createService(wsdlURL, name);

5  QName port = new QName("", "SOAPAccessPort");

6  Collie proxy = (Collie)service.getPort(port, Collie.class);
```

The code in Example 31 does the following:

1. Builds a UDDI URL to query the UDDI registry hosted at `localhost:9090` for services whose `tmodelname` is `collie`.
2. Builds a `QName` for the service proxy.
3. Gets an instance of the `ServiceFactory`.

4. Instantiates a new `Service` object using the endpoint information returned from the UDDI registry.
5. Builds a `QName` for the port that will be used to access the service.
6. Creates the service proxy.

Configuring your application to use UDDI support

The Artix UDDI support is provided by an Artix plug-in. To use the UDDI features, you must configure your application to load the Java version of the UDDI plug-in. To configure you application to load the UDDI plug-in do the following:

1. Open artix.cfg in any text editor.
2. Locate the scope for your application, or create a new one for it.
3. Add `java_uddi_proxy` to the list of plug-ins in the `java_plugins` list.
4. Add `java` to the list of plug-ins in the `orb_plugins` list.

Example 32 shows a configuration fragment with the configuration to use UDDI.

Example 32: *UDDI Configuration*

```
collieClient

{
  orb_plugins = ["java", "xmlfile_log_stream"];
  java_plugins = ["java_uddi_proxy"];
}
```

For more information on configuring Artix see Deploying and Managing Artix Solutions.

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 classloader firewall that isolates the Artix runtime classloader from the application classloader and the system classloader. 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 classloader firewall works

The classloader firewall provides a mechanism for you to hide the application classloader'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 classloader are visible to the Artix runtime's classloader and specifying the location from which the Artix runtime classloader will load its classes. Any classes not matched by a positive filter are blocked from the Artix runtime's classloader and will only be loaded from the locations specified in the firewall's configuration file. Figure 4 shows how the classloader firewall blocks off the Artix runtime.

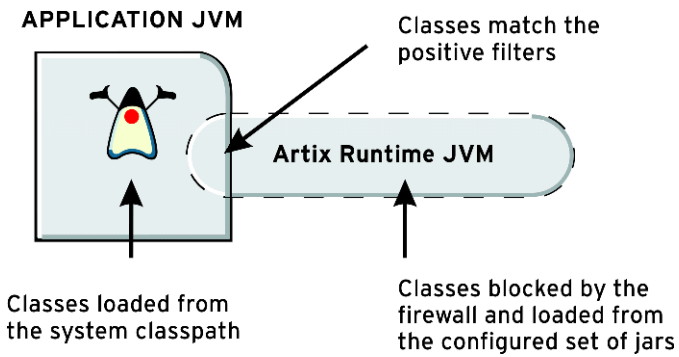

Figure 4: *Classloader Firewall*

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 classloader

To use the classloader 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 classloader to be visible to your application code.
3. Define the rules governing where the Artix classloader will look for specific classes in the `ce:loader` element of `artix_ce.xml`.

Defining class filters

The classloader 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 classloader. 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 classloader 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 Classloader 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 Classloader 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 classloader. 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 classloader 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 classloader 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 classloader 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 classloader 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 33 to specify the proper Xerces and Log4J version to load into the Artix runtime.

Example 33: *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 classloader firewall see the java_firewall demo in the demos\basic folder of your Artix installation. The demo provides an example of using the classloader 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:

XMLSchema Elements

Schema elements

Elements in XMLSchema represent an instance of an element in an XML document generated from the schema. At their most basic, an element consists of a single `element` element. Global `element` elements have two attributes:

- `name` specifies the name of the element as it will appear in an XML document.
- `type` specifies the type of the element. The type can be any XMLSchema primitive type or any named complex type defined in the contract.

In addition to `name` and `type`, global elements have one other commonly used optional attributes: `nillable`. This attribute specifies if an element can be left out of a document entirely. If `nillable` is set to `true`, the element can be omitted from any document generated using the schema.

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

Java mapping

Artix does not generate special classes for `element` elements unless they have an in-line type definition. For in-line type definitions Artix follows the same rules for code generation as described for a type definition. The mappings between XMLSchema types and Java classes is described in the following sections of this chapter.

Because Artix does not generate classes specifically for elements some of the attributes of XMLSchema elements are not supported. In particular, the attribute `"abstract=true"` is not recognized by Artix. If you specify that an element is abstract and give it an in-line type definition, Artix will still generate a class to support the defined type.

Using XMLSchema Simple Types

Overview

Artix follows the JAX-RPC specification for mapping native XMLSchema types into Java. In most cases, the mapping from an atomic 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:

Atomic Type Mapping

Overview

When a message part is described as being of one of the atomic XMLSchema types, the generated parameter's type will be of a corresponding primitive Java type. For example, the message description shown in Example 34 will cause a parameter, `score`, of type `int` to be generated.

Example 34: *Message Description Using a Simple Type*

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

Table of atomic type mappings

The atomics 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:normalizedString`	`java.lang.String`
`xsd:int`	`int`
`xsd:unsignedInt`	`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`

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

Schema Type	Java Type
xsd:unsignedByte	byte
xsd:integer	java.math.BigInteger
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:time	java.util.Calendar
xsd:date	java.util.Calandar
xsd:QName	javax.xml.namespace.QName
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:ID	java.lang.String
xsd:token	java.lang.String
xsd:language	java.lang.String
xsd:Name	java.lang.String
xsd:NCName	java.lang.String
xsd:NMTOKEN	java.lang.String
xsd:anySimpleType	java.lang.String
xsd:anyURI	java.net.URI
xsd:gYear	java.lang.String
xsd:gMonth	java.lang.String

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

Schema Type	Java Type
xsd:unsignedByte	byte
xsd:integer	java.math.BigInteger
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:time	java.util.Calendar
xsd:date	java.util.Calandar
xsd:QName	javax.xml.namespace.QName
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:ID	java.lang.String
xsd:token	java.lang.String
xsd:language	java.lang.String
xsd:Name	java.lang.String
xsd:NCName	java.lang.String
xsd:NMTOKEN	java.lang.String
xsd:anySimpleType	java.lang.String
xsd:anyURI	java.net.URI
xsd:gYear	java.lang.String
xsd:gMonth	java.lang.String

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

Schema Type	Java Type
`xsd:gDay`	`java.lang.String`
`xsd:gYearMonth`	`java.lang.String`
`xsd:gMonthDay`	`java.lang.String`

Atomic type validation

Artix Java validates XMLSchema atomic 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 atomics 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 Atomics Type Mappings

Overview

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

- an `element` element with its `nillable` attribute set to `true` as shown in Example 35.

  Example 35: *Nillable Element*

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

- an `element` 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 36.

  Example 36: *minOccurs set to Zero*

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

- an `attribute` element 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 37.

  Example 37: *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
xsd:unsignedByte	java.lang.Short
xsd:unsignedShort	java.lang.Integer
xsd:unsignedInt	java.lang.Long
xsd:unsignedLong	java.math.BigInteger

Defining Simple Types by Restriction

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 `type>` 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 38 shows the syntax for describing a simple type.

Example 38: *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 39 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 39: *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 39, would be mapped to a `String` because the base type of `SSN` is `xsd:string`. For example, the contract fragment shown in Example 40 would result in a Java method, `creditInfo()`, which took a parameter, `socNum`, of `String`.

Example 40: *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 "Using Enumerations" on page 71.

Using Enumerations

Overview

In XMLSchema, enumerations are described by derivation of a simple type using the syntax shown in Example 41.

Example 41: *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 42, would be valid if it were `<widgetSize>big</widgetSize>`, but not if it were `<widgetSize>big,mungo</widgetSize>`.

Example 42: *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 107.

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 43, to represent the enumeration, `widgetSize`, shown in Example 42 on page 71.

Example 43: *WidgetSize Class*

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

Example 43: *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 43: *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 43: *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 42 on page 71, 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 44 shows a simple text entry method for getting the proper member of the enumeration using `fromValue()`,

Example 44: *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 45 uses `getValue()` to return the value of the enumeration retrieved in Example 44 on page 75

Example 45: *Using getValue()*

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

Using 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 46.

Example 46: *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 47.

Example 47: *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 4 shows the facets used by list types.

Table 4: *List Type Facets*

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

Table 4: *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 46 on page 77, is shown in Example 48.

Example 48: *Definition for simpleList*

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

In addition to the syntax shown in Example 47 on page 77 you can also define a list type using the less common syntax shown in Example 49.

Example 49: *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 50 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 50: *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 51, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 51: *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>
...
```

Using XMLSchema Unions

Overview

In XMLSchema, a union is a construct that allows you to describe a type whose data can be of a number of simple types. For example, you could define a type whose value could be either the integer `1` or the string `first`.

XMLSchema unions are simple types, defined using a `simpleType` element. They contain at least one `union` element that define the *member types* of the union. The member types of the union are the valid types of data that can be stored in an instance of the union. You define them using the `memberTypes` attribute of the `union` element. `memberTypes` contains a list of one or more defined simple type names. Example 52 shows the definition of a union that can store either an integer or a string.

Example 52: *Simple Union Type*

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

In addition to specifying named types to be a member type of a union, you can also define anonymous simple types to be a member type of a union. This is done by adding the anonymous type definition inside of the `union` tag. Example 53 shows an example of a union containing an anonymous member type restricting the possible values of a valid integer to `1` through `10`.

Example 53: *Union with an Anonymous Member Type*

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
        <maxInclusive value="10" />
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Mapping to Java class

Artix maps unions to a Java class whose name is taken from the schema type's `name` attribute. So Artix would generate a class, `OrderNumUnion`, to represent the `orderNumUnion` union.

Note: If the union contains an anonymous enumerated type, the nested type will result in a generated class whose name begins with the name of the union and ends with the name of the base simple type. See "Using Enumerations" on page 71

The Java mapping of XMLSchema unions is very similar to that used in mapping choice complex types. See "Choice Complex Types" on page 90. The generated class would contain a getter method, a setter method and an `isSet` method for each member type in the union. For example, `orderNumUnion`, shown in Example 52 on page 80, would result in the generated class shown in Example 54.

Example 54: *Java Class for a Union*

```
public class OrderNumUnion
{
  private String __discriminator;
  private String string;
  private int _int

  public String getString()
  {
    return (String)string;
  }

  public setString(String val)
  {
    this.string = val;
    __discriminator = "string";
  }
```

Example 54: *Java Class for a Union*

```
  public boolean isSetString()
  {
    if(__discriminator != null &&
       __discriminator.eqauls("string"))
    {
      return true;
    }

  return false;
  }

  public get_int()
  {
    return (int)_int;
  }

  public set_int(int val)
  {
    this._int = val;
    __discriminator = "_int";
  }

  public boolean isSet_int()
  {
    if(__discriminator != null && _discriminator.eqauls("__int"))
    {
      return true;
    }

  return false;
  }

  public toString()
  {
  ...
  }
}
```

Working with unions in Java

When working with unions in Java it is important to remember that in XMLSchema only one of the member types can be valid at a time. This means that in an Artix Java application, while it is possible for both elements of the generated class can have valid data in them, only the last element on which `set` was called will be transmitted across the wire. For

example, if you called `set_int()` and then called `setString()`, both elements in `OrderNumUnion` would have valid data, but the discriminator would be set to the string member and that is the only value Artix will consider valid. If you transmitted the object, the receiving application would only receive the data stored in the string member.

Receiving union types in Artix is a little more complicated. When using bindings that pass information as XML documents, like SOAP, Artix will follow the validation rules described in the XMLSchema specification for determining the value of the union. So, if the `xsi:type` is written by the sending application, Artix will use that to determine the valid member element of the union. If the `xsi:type` is not written by the sending application, Artix will use the order in which the member types are specified in the type definition to determine the valid member type. For example, if an Artix application using a SOAP binding receives an element of type `OrderNumUnion` and the `xsi:type` is not written out by the sending application, the data will be treated as a string because `xsd:string` is first in the member type list.

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

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

Note: You can define a complex type without using `sequence`, `all`, or `choice`. However, the type can only contain attributes.

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 55, the generated setter method would have the signature `void setName(String val)`.

Example 55: *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 55 would have the signature `String getName()`.

Elements of `xsd:boolean` are an exception to the above mapping. For elements of type `xsd:boolean`, the getter methods name is prepended with is. For example if an element is defined as `<element name="in" type="xsd:boolean />` the generated getter method would be `boolean isIn()`.

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 56 would result in the generation of a private variable, `observedSpeed`,of type `float[]`.

Example 56: *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 57.

Example 57: *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 58.

Example 58: *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 59.

Example 59: *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;
  }

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

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

Example 59: *monsterStats Java Class*

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

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

  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 60.

Example 60: *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 61.

Example 61: *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 61: *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 61: *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` elements and `attributeGroup` elements within the scope of a `complexType` element. 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. For example, when describing the XML element `<value currency="euro">410<\value>` in XMLSchema `currency` would be described using an `attribute` element as shown in Example 62 on page 95.

The `attributeGroup` element allows you to define a group of reusable attributes that can be referenced by all complex types defined by the schema. For example, if you are defining a series of elements that all use the attributes `catagory` and `pubDate`, you could define an attribute group with these attributes and reference them in all the elements that use them. This is shown in Example 65 on page 96.

When describing data types for use in developing application logic, 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` element has one required attribute and three optional attributes. The `name` attribute is required and identifies the attribute. The `use` attribute specifies if the attribute is `required`, `optional`, or `prohibited`. 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. Example 62 shows an `attribute` element defining an attribute, `currency`, whose value is a string.

Example 62: *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>
```

If the `type` attribute is omitted from the `attribute` element, the format of the data value must be described as part of the `attribute` element. Example 63 shows an `attribute` element for an attribute, `catagory`, that can take the values `autobiography`, `non-fiction`, or `fiction`.

Example 63: *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 64 shows an alternate description of the `catagory` attribute using the `type` attribute.

Example 64: *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.

Describing an attribute group in XMLSchema

Using an attribute group in a complex type definition is a two step process. The first step is to define the attribute group itself. An attribute group is defined using an `attributeGroup` element with a number of `attribute` child elements. When defining an attribute group, `attributeGroup` requires a `name` attribute that defines the string used to refer to the attribute group. The `attribute` children elements define the members of the attribute group and are specified as shown in "Describing an attribute in XMLSchema" on page 94. Example 65 shows the description of the attribute group `catalogIndecies`. The attribute group has two members. `catagory` is of the type defined in Example 64 on page 96. `pubDate` is of the native XMLSchema type `dateTime` and is required.

Example 65: *Attribute Group Definition*

```
<attributeGroup name="catalogIndices">
  <attribute name="catagory" type="catagoryType" />
  <attribute name="pubDate" type="dateTime" use="required" />
</attributeGroup>
```

The second step is using an attribute group is to use the attribute group in the definition of a complex type. You use attribute groups in complex type definitions by using the `attributeGroup` element with the `ref` attribute. The value of the `ref` attribute is the name given the attribute group that you want to use as part of the type definition. For example if you wanted to use the attribute group `catalogIndecies` in the complex type `dvdType`, you would use `<attributeGroup ref="catalogIndecies" />` as shown in Example 66.

Example 66: *Complex Type with an Attribute Group*

```
<complexType name="dvdType">
  <sequence>
    <element name="title" type="xsd:string" />
    <element name="director" type="xsd:string" />
    <element name="numCopies" type="xsd:int" />
  </sequence>
  <attributeGroup ref="catalogIndecies" />
</complexType>
```

Mapping to Java

Attributes are mapped to elements in the generated Java class for a complex type. For each `attribute` element in a complex type definition, a corresponding element, along with getter and setter methods, will be added to the generated Java class for the type. For example, a contract with the complex type shown in Example 67 would generate a class with three sets of getter/setter methods.

Example 67: *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 it would be similar to Example 68.

Example 68: *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;
  }

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

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

Example 68: *techDoc Java Class*

```
  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();
  }
}
```

Attribute groups are mapped into Java as if the members of the group were explicitly used in the type definition. If your attribute group has three members, and it is used in a complex type, the generated class for that type will include an element, along with the getter and setter methods, for each member of the attribute group. For example, the complex type defined in Example 66, Artix would generate a class that contained the members `catagory` and `pubDate` to support the members of the attribute group used in the definition as shown in.

Example 69: *dvdType Java Class*

```
// Java
public class Dvd
{
  private String title;
  private String director;
  private short numCopies;
  private Catagory catagory;
  private Calendar pubDate;
```

Example 69: *dvdType Java Class*

```
public void setTitle(String val)
{
  title=val;
}
public String getTitle()
{
  return title;
}

public void setDirector(String val)
{
  director=val;
}
public String getDirector()
{
  return director;
}

public void setNumCopies(short val)
{
  numCopies=val;
}
public short getNumCopies()
{
  return numCopies;
}

public void setCatagory(Catagory val)
{
  catagory=val;
}
public Catagory getCatagory()
{
  return catagory;
}

public void setPubData(Calendar val)
{
  pubDate=val;
}
public Calendar getPubDate()
{
  return pubDate;
}
```

Example 69: *dvdType Java Class*

```
  public String 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 70.

Example 70: *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.

Artix will generate a class for each of the named types. The type containing the nested type will contain an element of the Java type generated for its class. For example, the type defined in Example 70 will result in the generation of two types:`TweetyBird` and `SylvesterState`. The generated type `SylvesterState` will contain an element food that is of type `TweetyBird`.

Example using named nested types

If you had an application using the complex type shown in Example 70 on page 102 your application would include two classes to support it, `TweetyBird` and `SylvesterState`.

Example 71 shows the generated Java class for `tweetyBird`.

Example 71: *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;
  }

  public int getGranny_proximity()
  {
    return granny_proximity;
  }

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

Example 71: *TweetyBird Class*

```
  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 72, has one element, `food`, that is an instance of `TweetyBird`.

Example 72: *SylvesterState Class*

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

  private int hunger;
  private TweetyBird food;

  public int getHunger()
  {
    return hunger;
  }

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

Example 72: *SylvesterState Class*

```
  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 73 shows an example of using the nested type `sylvesterState` in Java.

Example 73: *Working with Nested Complex Types*

```
// Java
SylvesterState hunter = new SylvesterState();
hunter.setHunger(25);

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

hunter.setFood(prey);
```

Example 73: *Working with Nested Complex Types*

```
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 73 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.

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 74 shows a description of `sylvesterState` using an anonymous type.

Example 74: *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.

When you use anonymous nested complex types, Artix generates a single class for the named complex type. The nested complex type is mapped to a public class that is internal to the generated class. The internal class will be given the name of the element for which it is generated. For example, the type defined in Example 74 would result in the generated class `SylvesterState`. The generated class `SylvesterState` contains a public class named `SylvesterState.Food` to represent the `food` element.

Example using anonymous nested types

If you had an application using the complex type shown in Example 71 on page 103 your application would include the class `SylvesterState` to support it.

The generated class for `sylvesterState`, shown in Example 75, contains an internal class `SylvesterState.Food`. The element `food` is an instance of `SylvesterState.Food`.

Example 75: *SylvesterState Class*

```
package com.iona.schemas.types.anoncattypes;

import java.util.Arrays;

public class SylvesterState
{
  public static final String TARGET_NAMESPACE =
   "http://schemas.iona.com/types/anonCatTypes";

  private int hunger;
  private Food food;

  public int getHunger()
  {
    return hunger;
  }

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

  public Food getFood()
  {
    return food;
  }

  public void setFood(Food val)
  {
    this.food = val;
  }
```

Example 75: *SylvesterState Class*

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

public static class Food
{
  public static final String TARGET_NAMESPACE =
 "http://schemas.iona.com/types/anonCatTypes";

  private boolean caged;
  private int granny_proximity;

  public boolean isCaged()
  {
    return caged;
  }

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

  public int getGranny_proximity()
  {
    return granny_proximity;
  }

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

Example 75: *SylvesterState Class*

```
    public String toString()
    {
      StringBuffer buffer = new StringBuffer();
      buffer.append("caged: "+caged+"\n");
      buffer.append("granny_proximity: "+granny_proximity+"\n");
      return buffer.toString();
    }
  }
}
```

When you set the value of `SylvesterState.food`, you must pass a valid `SylvesterState.Food` object to `setFood()`. Also, when you get the value of `SylvesterState.food`, you are returned a `SylvesterState.Food` object which has its own getter and setter methods. Example 73 shows an example of using the nested type `sylvesterState` in Java.

Example 76: *Working with Nested Complex Types*

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

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

3  hunter.setFood(prey);

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

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

The code in Example 73 does the following:

1. Instantiates a new `SylvesterState` object and sets its `hunger` element to 25.
2. Instantiates a new `SylvesterState.Food` 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 77 shows an example of a complex type, `internationalPrice`, derived by extension from the `xsd:decimal` simple type to include a currency attribute.

Example 77: *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` element 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 78 shows the generated Java class representing `internationalPrice` class generated from Example 77.

Example 78: *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.

Deriving a Complex Type from a Complex Type

Overview

Using XMLSchema, you can derive new complex types by extending or restricting 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 either the `extension` or the `restriction` element. The `complexContent` element specifies that the included data description includes more than one field. The `extension` element and the `restriction` element, which are part of the `complexContent` definition, specifies the base type being modified to create the new type. The base type is specified by the `base` attribute.

Extending a complex type

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 79 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 79: *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 79: *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>
```

Restricting a complex type

Within the `restriction` element you must list all of the elements and attributes of the base type. For each element you can add restrictive attributes to the definition. For example, you could add a `maxOccurs` attribute to an element to limit the number of times it can occur. You could also use the `fixed` attribute to force on or more of the elements to have predetermined values.

Example 80 shows an example of defining a complex type by restricting another complex type. The redefined type, `wallawallaAddress`, can only be used for addresses in Walla Walla, Washington because the values for `city`, `state`, and `zipCode` have been fixed.

Example 80: *Defining a Complex Type by Restriction*

```
<complexType name="Address">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="street" type="xsd:short" maxOccurs="3"/>
    <element name="city" type="xsd:string"/>
    <element name="state" type="xsd:string"/>
    <element name="zipCode" type="xsd:string"/>
  </sequence>
</complexType>
```

Example 80: *Defining a Complex Type by Restriction*

```
<complexType name="wallawallaAddress">
  <complexContent>
    <restriction base="xsd1:Address">
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="street" type="xsd:short" maxOccurs="3"/>
        <element name="city" type="xsd:string"
                 fixed="WallaWalla"/>
        <element name="state" type="xsd:string" fixed="WA" />
       <element name="zipCode" type="xsd:string"fixed="99362" />
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

Generated Java code

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

When the new complex type is derived by extension, the generated class will include getter and setter methods for all of the added elements and attributes. The new methods will be generated according to the same mappings as all other elements.

When the new complex type is derived by restriction, the generated class will have no new getter or setter methods. It will simply redefine the Artix specific information needed to marshal and unmarshal the data.

Note: Artix does not enforce the restriction defined in the contract. It is up to you to ensure that your application logic enforces them.

For example, the schema in Example 79 on page 115 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 81 shows the generated class for `widgetOrderBillInfo`.

Example 81: *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;
  }

  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();
  }
}
```

Occurrence Constraints

Overview

XMLSchema allows you to specify the occurrence constraints on three different XMLSchema elements that make up a complex type definition:

- The sequence element
- The choice element
- The element element

The sequence element

You can specify that a sequence of elements is to occur multiple times by setting the element's `minOccurs` and `maxOccurs` attributes. The `minOccurs` attribute specifies the minimum number of times the sequence must occur in an instance of the defined complex type. The `maxOccurs` attribute specifies the upper limit for how many times the sequence can occur in an instance of the defined complex type. Example 84 shows the definition of a sequence type, `CultureInfo`, with sequence occurrence constraints. The choice type overall can be repeated 0 to 2 times.

Example 82: *Sequence with Occurrence Constraints*

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

Mapping to Java

When a sequence with occurrence constraints is mapped into Java it looks very similar to a vanilla sequence. Each element still has a getter and setter methods. However, these methods all take an additional parameter, `index`, that specifies which instance of the sequence is being referenced. In addition, Artix generates a new internal sequence, *`TypeName`*`_Insternal`, and four new functions to cope with the multiple occurrences of the type:

- `_setSize()` allows you to specify how many times the sequence occurs.
- `_getSize()` returns the number of time the sequence occurs.

- `_set`*`TypeName`*`_Internal()` allows to set an instance of the sequence into one of the occurences.
- `_get`*`TypeName`*`_internal()` returns the instance of the sequence stored at the specified index.

Example 83 shows an outline of the Java class generated for the type defined in Example 82.

Example 83: *Java Class for Sequence with Occurrence Constraints*

```
public class CultureInfo
{
  private CultureInfo_Internal[] cultureInfo_Internal;

    public int _getSize() {
        if (null != cultureInfo_Internal) {
            return cultureInfo_Internal.length;
        }
        return 0;
    }
```

Example 83: *Java Class for Sequence with Occurrence Constraints*

```
 public void _setSize(int sz) {
     CultureInfo.CultureInfo_Internal[] temp = new
CultureInfo.CultureInfo_Internal[sz];
     if (null != cultureInfo_Internal) {
         if (sz <= cultureInfo_Internal.length) {
             for (int x = 0; x < sz; x++) {
                 temp[x] = cultureInfo_Internal[x];
             }
         } else {
             for (int x = 0; x < cultureInfo_Internal.length;
x++) {
                 temp[x] = cultureInfo_Internal[x];
             }
            for (int x = cultureInfo_Internal.length; x < sz;
x++) {
                 temp[x] = new
CultureInfo.CultureInfo_Internal();
             }
         }
     } else {
         for (int x = 0; x < sz; x++) {
             temp[x] = new CultureInfo.CultureInfo_Internal();
         }
     }
     cultureInfo_Internal = temp;
 }

 public void
_setCultureInfo_Internal(CultureInfo.CultureInfo_Internal
val, int indx) {
     this.cultureInfo_Internal[indx] = val;
 }

 public CultureInfo.CultureInfo_Internal
_getCultureInfo_Internal(int indx) {
     return cultureInfo_Internal[indx];
 }

 public void setName(java.lang.String val, int indx) {
     this.cultureInfo_Internal[indx].setName(val);
 }
```

Example 83: *Java Class for Sequence with Occurrence Constraints*

```
    public int getLcid(int indx) {
        return cultureInfo_Internal[indx].getLcid();
    }

    public void setLcid(int val, int indx) {
        this.cultureInfo_Internal[indx].setLcid(val);
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (cultureInfo_Internal != null) {
            buffer.append("cultureInfo_Internal : " +
   java.util.Arrays.asList(cultureInfo_Internal).toString() +
   "\n");
        }
        return buffer.toString();
    }

    public static class CultureInfo_Internal {

        private String name;
        private int lcid;

        public String getName() {
            return name;
        }

        public void setName(String val) {
            this.name = val;
        }


        public int getLcid() {
            return lcid;
        }

        public void setLcid(int val) {
            this.lcid = val;
        }
```

Example 83: *Java Class for Sequence with Occurrence Constraints*

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

The choice element

A choice type can also be defined with occurrence constraints.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 choice must occur in an instance of the defined complex type. The `maxOccurs` attribute specifies the upper limit for how many times the choice type can occur in an instance of the defined complex type. Example 84 shows the definition of a choice type, `ClubEvent`, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 84: *Choice Occurrence Constraints*

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

Mapping to Java

When a choice type with occurrence constraints is mapped into Java it looks very similar to a vanilla choice type. Each element still has a getter a setter and an `isSet` method. However, these methods all take an additional parameter, `index`, that specifies which instance of the choice type is being referenced. In addition, Artix generates a new internal choice type, *`TypeName`*`_Insternal`, and four new functions to cope with the multiple occurrences of the type:

- `_setSize()` allows you to specify how many times the choice type occurs.
- `_getSize()` returns the number of occurences of the choice type.
- `_set`*`TypeName`*`_Internal()` allows to set an instance of the choice type into one of the occurences.
- `_get`*`TypeName`*`_internal()` returns the instance of the choice type stored at the specified index.

Example 85 shows an outline of the Java class generated for the type defined in Example 84.

Example 85: *Java Class for Choice with Occurrence Constraints*

```
public class ClubEvent
{
  private String __discriminator;
  private ClubEvent_Internal[] clubEvent_Internal;

  public int _getSize()
  {
    if (null != clubEvent_Internal)
    {
      return clubEvent_Internal.length;
    }
    return 0;
  }
```

Example 85: *Java Class for Choice with Occurrence Constraints*

```
public void _setSize(int sz)
{
  ClubEvent.ClubEvent_Internal[] temp = new
 ClubEvent.ClubEvent_Internal[sz];
  if (null != clubEvent_Internal)
  {
    if (sz <= clubEvent_Internal.length)
     {
       for (int x = 0; x < sz; x++)
       {
         temp[x] = clubEvent_Internal[x];
       }
     }
     else
     {
       for (int x = 0; x < clubEvent_Internal.length; x++)
       {
         temp[x] = clubEvent_Internal[x];
       }
       for (int x = clubEvent_Internal.length; x < sz; x++)
       {
         temp[x] = new ClubEvent.ClubEvent_Internal();
       }
     }
  }
  else
  {
    for (int x = 0; x < sz; x++)
    {
      temp[x] = new ClubEvent.ClubEvent_Internal();
    }
  }
  clubEvent_Internal = temp;
}

public void _setClubEvent_Internal(
                ClubEvent.ClubEvent_Internal val,
                int indx)
{
  this.clubEvent_Internal[indx] = val;
}
```

Example 85: *Java Class for Choice with Occurrence Constraints*

```
public ClubEvent.ClubEvent_Internal _getClubEvent_Internal(
                                      int indx)
{
  return clubEvent_Internal[indx];
}

public java.lang.String getMemberName(int indx)
{
  return clubEvent_Internal[indx].getMemberName();
}

public void setMemberName(java.lang.String val, int indx)
{
  this.clubEvent_Internal[indx].setMemberName(val);
}

public boolean isSetMemberName(int indx)
{
  return clubEvent_Internal[indx].isSetMemberName();
}

public java.lang.String getGuestName(int indx)
{
  return clubEvent_Internal[indx].getGuestName();
}

public void setGuestName(java.lang.String val, int indx)
{
  this.clubEvent_Internal[indx].setGuestName(val);
}

public boolean isSetGuestName(int indx)
{
  return clubEvent_Internal[indx].isSetGuestName();
}
```

Example 85: *Java Class for Choice with Occurrence Constraints*

```
public String toString() {
    StringBuffer buffer = new StringBuffer();
    if (clubEvent_Internal != null) {
        buffer.append("clubEvent_Internal : " +
java.util.Arrays.asList(clubEvent_Internal).toString() +
"\n");
    }
    if (__discriminator != null) {
        buffer.append("Discriminator : " + __discriminator +
"\n");
    }
    return buffer.toString();
}

public static class ClubEvent_Internal {
    private String __discriminator;

    private String memberName;
    private String guestName;

    public String getMemberName() {
        return (String)memberName;
    }

    public void setMemberName(String val) {
        this.memberName = val;
        __discriminator = "memberName";
    }

    public boolean isSetMemberName() {
        if(__discriminator != null &&
           __discriminator.equals("memberName")) {
            return true;
        }
        return false;
    }
```

Example 85: *Java Class for Choice with Occurrence Constraints*

```
        public String getGuestName() {
            return (String)guestName;
        }

        public void setGuestName(String val) {
            this.guestName = val;
            __discriminator = "guestName";
        }

        public boolean isSetGuestName() {
            if(__discriminator != null &&
               __discriminator.equals("guestName")) {
                return true;
            }
            return false;
        }

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

The element element

You can set 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 86.

Example 86: *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 86, 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 144.

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 86 will be mapped to the Java class `HouseCat` shown in Example 87.

Example 87: *HouseCat Java Class*

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

Example 87: *HouseCat Java Class*

```
  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 Model Groups

Overview

XMLSchema model groups are a convenient shortcut that enables you to reference a group of elements from a user-defined complex type.For example, you could define a group of elements that are common to several types in your application and then reference the group repeatedly. Model groups are defined using the `group` element and are similar to complex type definitions. The mapping of model groups to Java is also similar to the mapping for complex types.

Defining a model group in XMLSchema

You define a model group in XMLSchema using the `group` element with the `name` attribute. The value of `name` is a string that is used to refer to the group throughout the schema. `group`, like `complexType`, can have either `sequence`, `all`, or `choice` as its immediate child element. Table 5 shows how the choice of child element affects the behavior of the elements in the group.

Table 5: *Group Children*

Child	Effect
`sequence`	All the members of the group must be present and are transmitted in the exact order they appear in the definition.
`all`	All of the members of the group must appear no more than once and their order in unimportant.
`choice`	No more than one member of the group can appear.

Inside the child element, you define the members of the group using `element` elements. For each member of the group, you specify one `element`. Group members can use any of the standard attributes for `element` including `minOccurs` and `maxOccurs`. So, if your group has three elements and one of

them can occur up to three times, you would define a group with three `element` elements, one of which would use `maxOccurs="3"`. Example 88 shows a model group with three elements.

Example 88: *Model Group*

```
<group name="passenger">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="clubNum" type="xsd:long" />
    <element name="seatPref" type="xsd:string" maxOccurs="3" />
  </sequence>
</group>
```

Using a model group in a type definition

Once a model group has been defined, you can use it as part of a complex type definition. To use a model group in a complex type definition, you use the `group` element with the `ref` attribute. The value of ref is the name given to the group when it was defined. For example, to use the group defined in Example 88 you would use `<group ref="tns:passenger" />` as shown in Example 89.

Example 89: *Complex Type with a Model Group*

```
<complexType name="reservation">
  <sequence>
    <group ref="tns:passenger" />
    <element name="origin" type="xsd:string" />
    <element name="destination" type="xsd:string" />
    <element name="fltNum" type="xsd:long" />
  </sequence>
</complexType>
```

When a model group is used in a type definition, the group becomes a member of the type. So an instance of `reservation` would have four members. The first of which would be `passenger` and have the members defined by the group in Example 88 as shown in Example 90.

Example 90: *Instance of a Type with a Group*

```
<reservation>
```

Example 90: *Instance of a Type with a Group*

```
  <passenger>
    <name>A. Smart</name>
    <clubNum>99</clubNum>
    <seatPref>isle1</seatPref>
  </passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

Mapping to Java

Artix maps model groups to Java classes using the same mapping used for complex types. For example, Artix would generate a Java class called `Passenger` to represent the group `passenger` defined in Example 88 on page 132. The generated class would have three members, one for each member of the group, and the associated getter and setter methods as shown in Example 91.

Example 91: *Class for a Group*

```
public class Passenger
{
  private String name;
  private long clubNum;
  private String[] seatPref;

  public String getName()
  {
    return name;
  }

  public void setName(String val)
  {
    this.name = val;
  }
```

Example 91: *Class for a Group*

```
  public long getClubNum()
  {
    return clubNum;
  }

  public void setClubNum(long val)
  {
    this.clubNum = val;
  }

  public String[] getSeatPref()
  {
    return seatPref;
  }

  public void setSeatPref(String[] val)
  {
    this.seatPref = val;
  }
}
```

If the group definition used `choice`, the Artix generated class would also include methods for determining which member of the group was valid. See "Using XMLSchema Complex Types" on page 84 for a detailed discussion of the mapping.

When Artix encounters a group in a complex type definition it maps the group to a class member of the type generated for the group's definition. For example, the generated class for reservation, defined in Example 89 on page 132, would include a member of type `Passenger` as shown in Example 92.

Example 92: *Type with a Group*

```
public class Reservation
{
    private Passenger passenger;
    private String origin;
    private String destination;
    private long fltNum;
```

Example 92: *Type with a Group*

```
  public Passenger getPassenger()
  {
    return passenger;
  }

  public void setPassenger(Passenger val)
  {
    this.passenger = val;
  }

  public String getOrigin()
  {
    return origin;
  }

  public void setOrigin(String val)
  {
    this.origin = val;
  }

  ...
}
```

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

Example 93: *Syntax of an any*

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

Table 6 explains the details of the optional attributes.

Table 6: *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 6: *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 correnspondes 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 94 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 94: *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 95 shows the Java class generated for the complex type `wildCard`, shown in Example 94 on page 138.

Example 95: *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 95: *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 necessarily need to determine the form of the data. You can still extract the contents using the `SOAPElement`'s methods.

Example 96 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 96: *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 97 shows code for extracting the data from an any containing a `houseCat`, defined in Example 86 on page 129.

Example 97: *Parsing the Contents of an any*

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

WildCard dataHolder;
```

Example 97: *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 97 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 98 shows the code for creating an any element containing the XML document `<houseCat><name>Joe</name><lives>12</lives></houseCat>`.

Example 98: *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 98 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 99.

Example 99: *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 100.

Example 100:*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 101 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 101: *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 102, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 102: *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 103, the mapping of `celebWasher()` would take a parameter, `badLang`, of type `String[][]`.

Example 103: *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.

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 104 shows an example of an operation that would require holder classes in the generated Java code.

Example 104: *Multiple Output Parts*

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

Example 104:*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 105 shows the resulting Java method signature.

Example 105:*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 106 shows how a server might implement the router method.

Example 106:*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 107 shows a client calling `router()` on a portal service.

Example 107:*Client Calling router()*

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

Example 107:*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 normally. 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 7 shows the mappings for the supported MIME types.

Table 7: *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 108 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 108: *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 108: *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` parameter, an `int` parameter, and a `byte[]` parameter. Instead the operation, `store`, is mapped as shown in Example 109.

Example 109: *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 attachments to a `DataHandler`, so the contract in Example 108 on page 152 would result in the operation shown in Example 110 as opposed to the one shown in Example 109 on page 154.

Example 110: *SOAP Attachments Using DataHandler Objects*

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

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

Example 110:*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 query 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.

Unsupported XMLSchema Constructs

Unsupported built-in types

The following XMLSchema types are currently not supported by Artix:

- `xsd:duration`
- `xsd:NOTATION`
- `xsd:IDREF`
- `xsd:IDREFS`
- `xsd:ENTITY`
- `xsd:ENTITIES`

Unsupported simpleType features

The following are not supported when working with `simpleType`:

- All facets except for `enumeration`
- The `final` attribute

Unsupported complexType features

The following are not supported when working with `complexType`:

- The `mixed` attribute
- The `final` attribute
- The `block` attribute
- The `abstract` attribute
- `simpleContent` with `restriction`

Unsupported attributes for element

The following attributes are not supported for `element`:

- `final`
- `block`
- `fixed`
- `default`
- `abstract`

Unsupported attributes for attribute

The following attributes are not supported for `attribute`:

- global attributes
- `ref`

- `from`

Unsupported group features

The following are not supported when working with `group`:

- `minOccurs` on local groups
- `maxOccurs` on local groups
- `all` inside a group

Other unsupported XMLSchema elements

The following XMLSchema elements are not supported:

- `xsd:redefine`
- `xsd:notation`
- `xsd:anyAttribute`
- `xsd:anySimpleType`
- `xsd:unique`
- `xsd:key`
- `xsd:keyref`
- `xsd:selector`
- `xsd:field`

id attribute

The `id` attribute is not supported by Artix.

CHAPTER 5

Using 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

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. Example 111

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

shows a message description in an Artix contract.

Example 111: *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` elements 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 112 shows an operation description that uses the message described in Example 111 on page 160 as a user-defined exception.

Example 112: *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 112, `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.

Mapping simple type exceptions

When your exception message is of a simple type, as shown in Example 111 on page 160, the generated type will have one private data member of the type specified in the contract's message part to represent the content of the message, a creation method that allows you to specify the values of the data member, and the associated getter and setter methods for the 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.

Mapping complex type exceptions

When your exception message is of a user defined complex type, Artix will generate an exception class whose name will be the name of the complex type used in the fault message postfixed with `_Exception`. For example, if you had a fault defined as shown in Example 113, the generated exception class would be named `NumInventory_Exception` and would be located in the same java package as the rest of the generated types.

Example 113:*Complex Fault*

```
...
<complexType name="numInventory">
  <sequence>
    <element name="numLeft" type="xsd:int" />
    <element name="size" type="xsd:string" />
  </sequence>
</complexType>
```

Example 113:*Complex Fault*

```
...
<message name="badSize">
  <part name="errorInfo" type="xsd1:numInventory" />
</message>
...
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
```

The generated exception class will be the same as the one generated for the complex type. The only difference being that the exception class extends `Exception` and is throwable. See "Working with Artix Data Types" on page 59.

Example

Example 114 shows the generated exception class for the fault message in Example 111 on page 160.

Example 114:*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 114:*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. Once the exception is caught, the client can inspect the contents using the standard methods.

Example

Example 115 shows how a server implementing the `getWidgets` operation, shown in Example 112 on page 161, might instantiate and throw a `NotEnoughInventory` exception.

Example 115: *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 116 shows how a client might catch and report the exception thrown by the server.

Example 116: *Catching a User-defined Exception*

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

Example 116:*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+".");
}
```

Working with CORBA Exceptions in Artix Applications

Overview

When integrating with CORBA services it is important to get an accurate picture of the exceptions that are returned. It is also important to ensure that proper CORBA exceptions are thrown back to the CORBA service. To make this possible Artix uses a Java class called `FaultException`. `FaultException` provides Artix Java applications access to basic CORBA exception handling capabilities.

In this section

This section discusses the following topics:

Mapping CORBA Exceptions to Artix Java Exceptions

Overview

By default, remote invocations in Java return a `RemoteException` when the remote service throws an exception. Artix can also return user-defined exceptions to the client. However, when making remote invocations on CORBA services, it is unlikely that the user has defined all of the possible exceptions that can be thrown and `RemoteException` objects do not provides enough detail to accurately determine the cause of the exception.

Artix services also have no way of throwing a CORBA exception back to CORBA clients. Artix services typically can only throw `RemoteException` or a user-defined exception. Native CORBA client code cannot do detailed error handling with these exceptions. All the client code knows is that something went wrong.

To fix this limitation, Artix uses a class called `com.iona.jbus.FaultException`. `FaultException` inherits from `RuntimeException` and adds fields to hold the information needed to support CORBA exceptions. Because they inherit from `RuntimeException`, `FaultException` objects can be thrown by Artix code and will be processed properly by the Artix runtime. You can also retirieve a `FaultException` object from the `RemoteException` object caught from a remove invocation. From the `FaultException` object you can get details about the CORBA exception.

FaultException fields

`FaultException` objects have four relevant fields. These field are explained in Table 8.

Table 8: *FaultException Fields*

Name	Description
`message`	The string used when creating the exception.
`Catagory`	The category of the exception. The category maps to the CORBA exception types. See Table 9 on page 169.
`CompletionStatus`	The status of the invocation. Maps to the CORBA completion status. See Table 10 on page 171.

Table 8: *FaultException Fields*

Name	Description
Source	The type of endpoint that threw the exception. Values are: • FaultSource.CLIENT • FaultSource.SERVER • FaultSource.UNKNOWN

CORBA Exceptions and Artix Fault Categories

Table 9 shows how each of the major CORBA system exceptions map to Artix fault categories.

Table 9: *Map from CORBA System Exceptions to Fault Categories*

CORBA System Exception	Fault Category
CORBA::BAD_CONTEXT	FaultCategory.INTERNAL
CORBA::BAD_INV_ORDER	FaultCategory.INTERNAL
CORBA::BAD_OPERATION	FaultCategory.BAD_OPERATION
CORBA::BAD_TYPECODE	FaultCategory.MARSHAL_ERROR
CORBA::BAD_QOS	FaultCategory.INTERNAL
CORBA::CODESET_INCOMPATIBLE	FaultCategory.MARSHAL_ERROR
CORBA::COMM_FAILURE	FaultCategory.CONNECTION_FAILURE
CORBA::DATA_CONVERSION	FaultCategory.MARSHAL_ERROR
CORBA::FREE_MEM	FaultCategory.MEMORY
CORBA::IMP_LIMIT	FaultCategory.INTERNAL
CORBA::INITIALIZE	FaultCategory.UNKNOWN
CORBA::INTERNAL	FaultCategory.INTERNAL
CORBA::INTF_REPOS	FaultCategory.INTERNAL
CORBA::INV_FLAG	FaultCategory.INTERNAL

Table 9: *Map from CORBA System Exceptions to Fault Categories*

CORBA System Exception	Fault Category
CORBA::INV_IDENT	FaultCategory.NOT_EXIST
CORBA::INV_OBJREF	FaultCategory.INVALID_REFERENCE
CORBA::INV_POLICY	FaultCategory.INTERNAL
CORBA::INVALID_TRANSACTION	FaultCategory.INTERNAL
CORBA::MARSHAL	FaultCategory.MARSHAL_ERROR
CORBA::NO_IMPLEMENT	FaultCategory.NOT_IMPLEMENTED
CORBA::NO_MEMORY	FaultCategory.MEMORY
CORBA::NO_PERMISSION	FaultCategory.NO_PERMISSION
CORBA::NO_RESOURCES	FaultCategory.INTERNAL
CORBA::NO_RESPONSE	FaultCategory.INTERNAL
CORBA::OBJ_ADAPTER	FaultCategory.INTERNAL
CORBA::OBJECT_NOT_EXIST	FaultCategory.NOT_EXIST
CORBA::PERSIST_STORE	FaultCategory.INTERNAL
CORBA::REBIND	FaultCategory.INTERNAL
CORBA::TIMEOUT	FaultCategory.TIMEOUT
CORBA::TRANSACTION_MODE	FaultCategory.INTERNAL
CORBA::TRANSACTION_REQUIRED	FaultCategory.INTERNAL
CORBA::TRANSACTION_ROLLEDBACK	FaultCategory.INTERNAL
CORBA::TRANSACTION_UNAVAILABLE	FaultCategory.INTERNAL
CORBA::TRANSIENT	FaultCategory.TRANSIENT

Completion status mapping

Table 10 shows the mapping between CORBA completion status values and fault completion status values.

Table 10: *Completion Status Mapping*

CORBA Completion Status	Fault Completion Status
`CORBA::COMPLETED_YES`	`FaultCompletionStatus.YES`
`CORBA::COMPLETED_NO`	`FaultCompletionStatus.NO`
`CORBA::COMPLETED_MAYBE`	`FaultCompletionStatus.MAYBE`

Throwing CORBA Exceptions from Artix

Overview

Simulating a CORBA exception from Artix Java code is a five step process:

1. Instantiate a `FaultException` object to hold the exception.
2. Set the exception's category field.
3. Set the exception's source field.
4. Set the exception's completion status field.
5. Throw the exception.

Instantiating a FaultException object

The `FaultException` class' creator method, shown in Example 117, takes a single string that is placed in the message field of the new object.

Example 117: *FaultException Creators*

```
FaultException(String message)
```

While it is good practice to populate the message field with a message describing the nature of the exception, it is not required.

None of the fields in the newly instantiated `FaultException` object will be initialized. You will need to set values for each field independently.

Setting the FaultException object's fields

`FaultException` objects have three setter methods, shown in Example 118, to populate the fields used to support CORBA exceptions.

Example 118: *FaultException Setter Methods*

```
void setCategory(FaultCategory faultCategory)
void setCompletionStatus(FaultCompletionStatus faultStatus)
void setSource(FaultSource faultSource)
```

The values used to set the categories are defined as enumerations, so the easiest way to set the values is to use the a static instance of the appropriate class. For example to set the source field to `UNKNOWN` you could use the code shown in Example 119.

Example 119: *Setting the Source Field*

```
fe.setSource(FaultSource.UNKNOWN);
```

Throwing a FaultException

The `FaultException` class is a child of the Java `RuntimeException` class. It can be thrown using the Java `throw` method. The Artix runtime will process the `FaultException` and create an appropriate fault message to place on the wire.

The recipient of the exception will receive a fault message that is appropriate for its binding. If the recipient is a native CORBA application it will receive a completely populated CORBA system exception. An Artix endpoint using a CORBA binding can decode the `FaultException` object if needed. Endpoints using other bindings will not get the CORBA information.

Example

Example 120 shows code for throwing a `CORBA::TRANSIENT` exception from an Artix service.

Example 120: *Throwing a CORBA Exception from Artix*

```
1 FaultException fe = new FaultException("Account has expired");
2 fe.setCategory(FaultCategory.NO_PERMISSION);
3 fe.setSource(FaultSource.SERVER);
4 fe.setCompletionStatus(FaultCompletionStatus.NO);
5 throw fe;
```

Processing CORBA Exceptions

Overview

If your code may need to catch CORBA exceptions, you can do the following:

1. Catch the `RemoteException` as normal.
2. Extract the chained cause of the `RemoteException` object using the `getCause()` method.
3. Check if the `Throwable` object is an instance of the `FaultException` class.
4. If it is, cast the `Throwable` object into a `FaultException` object.
5. Use the `FaultException` object's get methods to extract the information about the CORBA exception.

Getting exception details from a FaultException

`FaultException` objects have three getter methods, shown in Example 121, to retrieve the information about a CORBA exception.

Example 121: *FaultException Getter Methods*

```
FaultCategory getCategory()
FaultCompletionStatus getCompletionStatus()
FaultSource getSource()
```

Evaluating the exception data

The values returned by the methods are instances of an enumeration, so the easiest way to evaluate the values is to use the a static instance of the appropriate class. For example to decide how to proceed based on the completion status you could use the code shown in Example 122.

Example 122: *Evaluating the Completion Status*

```
    FaultCompletionStatus fcs = fe.getCompletionStatus();
    if (fcs.value().equals(FaultCompletionStatus.YES)
    {
      // Operation completed
    }
    else
    {
      // Operation not completed
    }
```

Example

Example 123 shows code for processing a CORBA exception in Artix.

Example 123: *Processing a CORBA exception in Artix*

```
try
{
  ....
  Client client = (Client)service.getPort(...);
  client.sayHi();
}
catch (RemoteException re)
{
  Throwable t = re.getCause();

  if (t instanceof FaultException)
  {
    FaultException fe = (FaultException) t;

    FaultCategory fc = fe.getCategory();
    if (fc.value().equals(FaultCategory.TRANSIENT)
    {
      // a CORBA TRANSIENT system exception
    }
```

Example 123:*Processing a CORBA exception in Artix*

```
    FaultCompletionStatus fcs = fe.getCompletionStatus();
    if (fcs.value().equals(FaultCompletionStatus.YES)
    {
      // Operation completed
    }

    FaultSource fs = fe.getSource();
    if (fs.value().equals(FaultSource.UNKNOWN)
    {
      // The exception was thrown by an unidentified endpoint
    }
  }

...
}
```

CHAPTER 6

Using Substitution Groups

XMLSchema substitution groups allow you to define a group of elements that can replace a top level, or head, element.

In this chapter

This chapter discusses the following topics:

Substitution Groups in XML Schema

Overview

A substitution group is a feature of XML schema that allows you to specify elements that can replace another element in documents generated from that schema. The replaceable element is called the head element and must be defined in the schema's global scope. The elements of the substitution group must be of the same type as the head element or a type that is derived from the head element's type.

In essence, a substitution group allows you to build a collection of elements that can be specified using a generic element. For example, if you are building an ordering system for a company that sells three types of widgets you may define a generic widget element that contains a set of common data for all three widget types. Then you could define a substitution group that contains a more specific set of data for each type of widget. In your contract you could then specify the generic widget element as a message part instead of defining a specific ordering operation for each type of widget. When the actual message is built, the message can then contain any of the elements of the substitution group.

Syntax

Substitution groups are defined using the `substitutionGroup` attribute of the XMLSchema `element` element. The value of the `substitutionGroup` attribute is the name of the element that the element being defined can replace. For example if your head element was `widget`, then by adding the attribute `substitutionGroup="widget"` to an element named `woodWidget` would specify that anywhere `widget` was used, you could substitute `woodWidget`. This is shown in Example 124.

Example 124: *Using a Substitution Group*

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
         substitutionGroup="widget" />
```

Type restrictions

The elements of a substitution group must be of a similar type to the head element of the group. This means that all the elements of the group must be of the same type as the head element or of a type derived from the head

element's type. For example, if the head element is of type `xsd:int` all members of the substitution group must be of type `xsd:int` or of type derived from `xsd:int`. You could also define a substitution group similar to the one shown in Example 125 where the elements of the substitution group are of types derived from the head element's type.

Example 125: *Substitution Group with Complex Types*

```
<complexType name="widgetType">
  <sequence>
    <element name="shape" type="xsd:string" />
    <element name="color" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="woodWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="woodType" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="plasticWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="moldProcess" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="widget" type="widgetType" />
<element name="woodWidget" type="woodWidgetType"
         substitutionGroup="widget" />
<element name="plasticWidget" type="plasticWidgetType"
         substitutionGroup="widget" />
<complexType name="partType">
  <sequence>
    <element ref="widget" />
  </sequence>
</complexType>
<element name="part" type="partType" />
```

The head element of the substitution group, `widget`, is defined as being of type `widgetType`. Each element of the substitution group then extends `widgetType` to include data specific to ordering the specific type of widget.

Based on the schema in Example 125 on page 179, the `<part>` elements in Example 126 are valid.

Example 126: *XML Document using a Substitution Group*

```
<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
    <color>blue</color>
    <moldProcess>sandCast</moldProcess>
  </plasticWidget>
</part>
<part>
  <woodWidget>
    <shape>round</shape>
    <color>blue</color>
    <woodType>elm</woodType>
  </woodWidget>
</part>
```

Abstract head elements

You can define an abstract head element that can never appear in a document produced using your schema. Abstract head elements are similar to abstract classes in Java in that they are used as the basis for defining more specific implementations of a generic class. Abstract heads also prevent the use of the generic element in the final product.

You declare an abstract head element using the `abstract="true"` attribute of `element` element as shown in Example 127. Using this schema, a valid `review` element could contain either a `positiveComment` element or a `negativeComment` element, but not a `comment` element.

Example 127:*Abstract Head Definition*

```
<element name="comment" type="xsd:string" abstract="true" />
```

Example 127:*Abstract Head Definition*

```
<element name="positiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="negtiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="review">
  <complexContent>
    <all>
      <element name="custName" type="xsd:string" />
      <element name="impression" ref="comment" />
    </all>
  </complexContent>
</element>
```

Using Substitution Groups with Artix

Overview

Artix allows you to use substitution groups when defining Artix systems. The bus properly validates messages that contain substitution groups provides a Java mapping that makes using a substitution group easy. Artix maps substitution groups into Java classes that extend the class used to represent the head class. In addition, it adds special getter and setter methods to complex types that reference members of substitution groups. Therefore, your application code can reflect the element hierachy defined in the WSDL.

Using a substitution group as an element of a complex type

When you include the head element of a substitution group as an element in a complex type, the Artix WSDL to Java code generator adds additional methods to the generated class representing the complex type. These methods are similar to the ones generated to support `choice` complex types. They allow you to place one of the elements of the substitution group into the object, query the object to determine which element of the substitution group is present in the object, and get a type specific element of the substitution group back from the object.

SImilar to how Artix generates code for `choice` complex types, Artix generates three methods for each element of a substitution group used in a complex type. These methods are a setter method named `set`*`MemberName`*`()`, a getter method named `get`*`MemberName`*`()`, and a method to determine if the element is the one being used by the object named `isSet`*`MemberName`*`()`. When setting a value into the object, you should use the element specific methods to ensure that the Artix runtime handles the data correctly when transmitting it across the wire.

For example, you could define a complex type named `widgetOrderInfo` that included an element defined using the widget element in Example 125 on page 179. A possible definition `widgetOrderInfo` is shown in Example 128.

Example 128: *Complex Type with a Substitution Group*

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:boolean" use="optional" />
</complexType>
```

Artix would generate the class shown in Example 129 to represent `widgetOrderInfo`. Unlike the other elements in the generated class, which only have a getter and a setter method, the `widget` element results in the generation of the methods `setWidget()`, `getWidget()`, `isSetWidget()`, `setWoodWidget()`, `getWoodWidget()`, `isSetWoodWidget()`, `setPlasticWidget()`, `getPlasticWidget()`, and `isSetPlasticWidget()` to handle the substitution group. However, like all of the other elements, the `widget` element only results in one member of the generated class. This member, `widget`, is of the type generated for the head element of the substitution group, `WidgetType`. This is possible because the types for each member of the substitution group inherit from `WidgetType`.

While, due to the inheritance rules in Java, you could use the generic `setWidget()` and `getWidget()` methods to place any one of the substitution group elements into the object, it is not advisable. Artix relies on the discriminator that is set in the type specific setter methods to ensure that

messages are generated properly when they are sent on the wire. So setting a `PlasticWidget` using `setWidget()` may produce unpredictable results in a running system.

Example 129:*Class for a Substitution Group*

```
public class WidgetOrderInfo
{
    private String __discriminator_widget;

    private int amount;
    private WidgetType widget;
    private Address shippingAddress;
    private Boolean rush;

    public int getAmount() {
        return amount;
    }

    public void setAmount(int val) {
        this.amount = val;
    }

    public WidgetType getWidget() {
        return widget;
    }

    public void setWidget(WidgetType val) {
        this.widget = val;
        __discriminator_widget = "widget";
    }

    public boolean isSetWidget() {
        if(__discriminator_widget != null &&
           __discriminator_widget.equals("widget")) {
            return true;
        }
        return false;
    }
```

Example 129:*Class for a Substitution Group*

```
public WoodWidgetType getWoodWidget() {
    return (WoodWidgetType)widget;
}

public void setWoodWidget(WoodWidgetType val) {
    this.widget = val;
    __discriminator_widget = "woodWidget";
}

/**
 * isSetWoodWidget
 *
 * @return: boolean
 */
public boolean isSetWoodWidget() {
    if(__discriminator_widget != null &&
       __discriminator_widget.equals("woodWidget")) {
        return true;
    }
    return false;
}

public PlasticWidgetType getPlasticWidget() {
    return (PlasticWidgetType)widget;
}

public void setPlasticWidget(PlasticWidgetType val) {
    this.widget = val;
    __discriminator_widget = "plasticWidget";
}

public boolean isSetPlasticWidget() {
    if(__discriminator_widget != null &&
       __discriminator_widget.equals("plasticWidget")) {
        return true;
    }
    return false;
}
```

Example 129:*Class for a Substitution Group*

```
    public Address getShippingAddress() {
        return shippingAddress;
    }

    public void setShippingAddress(Address val) {
        this.shippingAddress = val;
    }

    public Boolean isRush() {
        return rush;
    }

    public void setRush(Boolean val) {
        this.rush = val;
    }

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

If the head element of the substitution group is declared abstract, the generated class will not include the methods to support the head element. So in Example 129, `getWidget()`, `setWidget()`, and `isSetWidget()` would not be generated.

Using a substitution group as an argument to an operation

When you use a substitution group as part of an operation's message, the Artix WSDL to Java code generator generates the method for the operation normally. The message part that is a substitution group results in a

parameter of the head element's type. For example, you could define the operation shown in Example 130 that uses the substitution group defined in Example 125 on page 179.

Example 130: *Operation with a Substitution Group*

```
<message name="widgetMessage">
    <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

Artix would generate the interface shown in Example 131 to implement `orderWidgets`. You could invoke on this operation by passing any of the valid elements of the widget substitution group as a parameter.

Example 131: *orderWidgets Generated Code*

```
public interface OrderWidgets extends java.rmi.Remote
{
  public int checkWidgets(
        com.widgetvendor.types.widgettypes.WidgetType widgetPart)
  throws RemoteException;
}
```

Because Artix generates the same code for elements and types, Artix does not enforce the `abstract` attribute when you use the head element of a substitution group as a message part. If you want to ensure that the

abstract attribute is enforced you should define a new element that includes a reference to the substitution group's head element and use that in place of the head element. This is shown in Example 132.

Example 132:*Element Referring to a Substitution Group*

```
<types ...>
...
  <element name="widgetElement">
    <complexType>
      <sequence>
        <element ref="xsd1:widget" />
      </sequence>
    </complexType>
  </element>
...
</types>
<message name="widgetMessage">
  <part name="request" element="xsd1:widgetElement" />
</message>
<message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

Doing so will cause Artix to generate a new class for the element that includes the appropriate methods for working with a substitution group. The generated method will use the class generated for the new element. The additional code generated to implement the contract fragment in Example 132 is shown in Example 133. In this scenario, if the head element is declared abstract the methods supporting it would not be generated.

Example 133:*Code for Element with a Substitution Group*

```
public class WidgetElement
{
    private String __discriminator_widget;

    private WidgetType widget;

    public WidgetType getWidget()
    {
        return widget;
    }

    public void setWidget(WidgetType val)
    {
        this.widget = val;
        __discriminator_widget = "widget";
    }

    public boolean isSetWidget()
    {
        if(__discriminator_widget != null &&
           __discriminator_widget.equals("widget")) {
            return true;
        }
        return false;
    }
```

Example 133:*Code for Element with a Substitution Group*

```
public WoodWidgetType getWoodWidget()
{
    return (WoodWidgetType)widget;
}

public void setWoodWidget(WoodWidgetType val)
{
    this.widget = val;
    __discriminator_widget = "woodWidget";
}

public boolean isSetWoodWidget()
{
    if(__discriminator_widget != null &&
       __discriminator_widget.equals("woodWidget")) {
        return true;
    }
    return false;
}
```

Example 133:*Code for Element with a Substitution Group*

```
    public PlasticWidgetType getPlasticWidget()
    {
        return (PlasticWidgetType)widget;
    }

    public void setPlasticWidget(PlasticWidgetType val)
    {
        this.widget = val;
        __discriminator_widget = "plasticWidget";
    }

    public boolean isSetPlasticWidget()
    {
        if(__discriminator_widget != null &&
           __discriminator_widget.equals("plasticWidget")) {
            return true;
        }
        return false;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        if (widget != null) {
            buffer.append("widget: "+widget+"\n");
        }
        return buffer.toString();
    }
}
public interface OrderWidgets extends java.rmi.Remote
{
  public int checkWidgets(
    com.widgetvendor.types.widgettypes.WidgetElement widgetPart)
  throws RemoteException;
}
```

Widget Vendor Example

Overview

This section shows an example of substitution groups being used in Artix to solve a real world application. A server and client are developed using the `widget` substitution group defined in Example 125 on page 179. The service offers two operations: `checkWidgets` and `placeWidgetOrder`. Example 134 shows the interface for the ordering service.

Example 134: *Widget Ordering Interface*

```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
</message>
<message name="widgetMessage">
  <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

The type `widgetOrderForm` is defined in Example 128 and `widgetOrderBillInfo` extends `widgetOrderForm` to include one extra field to hold the final cost of the order.

Note: Because the example is to demonstrate the use of substitution groups, some of the business logic is not shown.

placeWidgetOrder

`placeWidgetOrder` takes a complex type containing the substituion group and then returns a complex type that contains a complex type. This operation demonstrates how one might go about using such a structure in a Java implentation. Both the client and the server have to get and set members of a substitution group.

checkWidgets

`checkWidgets` is a simple operation that has a parameter that is a substitution group. This operation demonstrates how to deal with individual parameters that are members of a substitution group. The server must properly determine which member of the substitution group was sent in the request. The client must ensure that the parameter is a valid member of the substitution group.

In this section

This section discusses the following topics:

Widget Server

Overview

The widget server implements the operations defined by the `orderWidgets` interface shown in Example 134. The Artix WSDL to Java code generator creates the implementation class shown in Example 135 for the interface. Using this as a starting point, the following section implements each of the defined operations. Note that some of the application logic is omitted for clarity around the use of substitution groups.

Example 135: *Widget Server Implementation Class*

```
// Java
package com.widgetvendor.widgetorderform;

import com.widgetvendor.types.widgettypes.WidgetOrderBillInfo;
import com.widgetvendor.types.widgettypes.WidgetOrderInfo;
import com.widgetvendor.types.widgettypes.WidgetType;

public class OrderWidgetsImpl implements java.rmi.Remote
{

  public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo
         placeWidgetOrder(com.widgetvendor.types.widgettypes.WidgetOrderInfo widgetOrderForm)
  {
    // User code goes in here.
    return new com.widgetvendor.types.widgettypes.WidgetOrderBillInfo();
  }

  public int checkWidgets(com.widgetvendor.types.widgettypes.WidgetType widgetPart)
  {
    // User code goes in here.
    return 0;
  }
}
```

placeWidgetOrder

`placeWidgetOrder()` recieves an order in the form of a `WidgetOrderInfo` object, processes the order, and returns a bill to the client in the form of a `WidgetOrderBillInfo` object. The orders can be for either a plain widget, a plastic widget, or a wooden widget. The type of widget ordered is

determined by what type of object is stored in `widgetOrderForm`'s `widget` member. `widget` is a substitution group and can contain either a `Widget`, a `WoodWidget`, or a `PlasticWidget`.

The best way to determine the type of object stored in `widgetOrderForm`'s `widget` member is to use the `isSet`*`elemName()`* methods. These methods are generated by the Artix WSDL to Artix code generator to support the identification of which element of a substitution group is being used and return a boolean value. Using these methods, you can build a series of if/then statements to determine what type of widget is being ordered and process the order correctly. This is shown in Example 136.

Example 136: *placeWidgetOrder()*

```
//Java
public WidgetOrderBillInfo placeWidgetOrder(WidgetOrderInfo
   widgetOrderForm)
{
  WidgetOrderBillInfo bill = new WidgetOrderBillInfo()

   // Copy the shipping address and the number of widgets
   // ordered from widgetOrderForm to bill
   ...
  int numOrdered = widgetOrderForm.getAmount();

  if (widgetOrderForm.isSetWidget())
  {
    // Get the widget data from the order form
    WidgetType order = widgetOrderForm.getWidget();

    // Method buildWidget() is left for you to implement
    buildWidget(order, numOrdered);

    // Add the amount of the bill and the widget info to bill
    bill.setWidget(order);
    float amtDue = numOrdered * 0.30;
    bill.setAmountDue(amtDue);
  }
```

Example 136:*placeWidgetOrder()*

```
  else if (widgetOrderForm.isSetWoodWidget())
  {
    // Get the widget data from the order form
    WoodWidgetType order = widgetOrderForm.getWoodWidget();

    // Method buildWoodWidget() is left for you to implement
    buildWoodWidget(order, numOrdered);

    // Add the amount of the bill and the widget info to bill
    bill.setWoodWidget(order);
    float amtDue = numOrdered * 0.85;
    bill.setAmountDue(amtDue);
  }
  else if (widgetOrderForm.isSetPlasticWidget())
  {
    // Get the widget data from the order form
    PlasticWidgetType order = widgetOrderForm.getPlasticWidget();

    // Method buildPlasticWidget() is left for you to implement
    buildPlasticWidget(order, numOrdered);

    // Add the amount of the bill and the widget info to bill
    bill.setPlasticWidget(order);
    float amtDue = numOrdered * 0.85;
    bill.setAmountDue(amtDue);
  }

  return bill;
}
```

Once you have determined which type of widget is in the order, you use the type specific getter method to extract the proper element of the substitution group in the order. To set the `widget` member of the bill you use the type specific setter methods to ensure that when the client gets the bill back it can use the `isSet`*`elemName()`* methods on the bill.

---

checkWidgets

`checkWidgets()` gets a widget description as a `WidgetType`, checks the inventory of widgets, and returns the number of widgets in stock. Due to the way Artix generates code, the fact that the operation is defined using a substitution group head element does not imply that you need to use any Artix specific APIs. In fact, you can implement `checkWidgets()` using standard Java code.

Because all of the types defining the different members of the substitution group for `widget` extend `WidgetType`, you can use `instanceof` to determine what type of widget was passed in and simply cast the argument `widgetPart` into the more restrictive type if appropriate. Once you have the proper type of object, you can check the inventory of the right kind of widget.

A possible implementation is shown in Example 137.

Example 137: *checkWidgets()*

```
public int checkWidgets(WidgetType widgetPart)
{
  if (widgetPart instanceof WidgetType)
  {
    return checkWidgetInventory(widgetType);
  }
  else if (widgetPart instanceof WoodWidgetType)
  {
    WoodWidgetType widget = (WoodWidgetType)widgetPart;
    return checkWoodWidgetInventory(widget);
  }
  else if (widgetPart instanceof PlasticWidgetType)
  {
    PlasticWidgetType widget = (PlasticWidgetType)widgetPart;
    return checkPlasticWidgetInventory(widget);
  }
}
```

Widget Client

Overview

The widget client makes request on the widget server for orders or to check inventory. To do so it must properly populate the data elements that are defined using substitution groups. For example, to make an order the client needs to use the type specific setter methods for the widget type it is ordering.

placeWidgetOrder

To invoke `placeWidgetOrder()` the client needs to construct a widget order that contains one element of the widget substitution group. When adding the widget to the order, the client code should use the type specific setters generated for each element of the substitution group to ensure that the Artix runtime and the server can correctly process the order. For example, if an order is being placed for a plastic widget, `setPlasticWidget()` should be used to add the widget to the order.

Example 138 shows client code for setting the `widget` member of `WidgetOrderInfo`.

Example 138: *Setting a Substitution Group Member*

```
//Java
InputStreamReader inReader = new InputStreamReader(System.in);
BufferedReader reader = new BufferedReader(inReader);

WidgetOrderInfo order = new WidgetOrderInfo();
...

System.out.println();
System.out.println("What color widgets do you want to order?");
String color = reader.readLine();
System.out.println();
System.out.println("What shape widgets do you want to order?");
String shape = reader.readLine();
```

Example 138:*Setting a Substitution Group Member*

```
System.out.println();
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);

switch (widgetType)
{
  case '1':
  {
    WidgetType widget = new WidgetType();
    widget.setColor(color);
    widget.setShape(shape);
    order.setWidget(widget);
    break;
  }
  case '2':
  {
    WoodWidgetType woodWidget = new WoodWidgetType();
    woodWidget.setColor(color);
    woodWidget.setShape(shape);

    System.out.println();
    System.out.println("What type of wood are your widgets?");
    String wood = reader.readLine();
    woodWidget.setWoodType(wood);

    order.setWoodWidget(woodWidget);
    break;
  }
```

Example 138:*Setting a Substitution Group Member*

```
  case '3':
  {
    PlasticWidgetType plasticWidget = new PlasticWidgetType();
    plasticWidget.setColor(color);
    plasticWidget.setShape(shape);

    System.out.println();
    System.out.println("What type of mold to use for your
                        widgets?");
    String mold = reader.readLine();
    plasticWidget.setMoldProcess(mold);

    order.setPlasticWidget(plasticWidget);
    break;
  }
  default :
    System.out.println("Invaid Widget Selection!!");
}
```

checkWidgets

Because substitution groups are made up of elements that are either of the same type or of element whose type inherits from the type of the head element, the client can invoke `checkWidgets()` without using any special Artix code. When developing the logical to invoke `checkWidgets()` you can pass in any element of the widget substitution group and the server side implementation should be able to handle it correctly.

The only caveat is that Artix does not enforce `abstract="true"`. It is up to you to ensure that your code does not pass in the head element in this case. This is particularly important when working with services that were not developed using Artix.

CHAPTER 7

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

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. Edit the WSDL path hard coded into the generated type factory to point to the proper location of your application's contract.
3. 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.

The generated type factories have a hard coded WSDL path. The WSDL path in the generated type factory is an absolute path that points to the location of the document from which the type factory was generated. If you plan to move your application, you will need to edit this hard coded path.

Java packages for type factory 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 42.
2. Instantiate the type factories you wish to register with the client proxy as shown in "Instantiating a type factory" on page 204.
3. Register the type factories using `registerTypeFactory()` on the `Bus` object as shown in "Registering a type factory" on page 205.

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 139 shows the code to instantiate the type factory for `packageDepot`.

Example 139: *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 140 shows code registering a type factory.

Example 140: *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 141 shows code for returning the hash table of registered type factories.

Example 141: *Getting Hash Table of Registered Type Factories*

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

Example

Example 142 shows an example of registering two type factories, `packageDepotTypeFactory` and `widgetsTypeFactory`.

Example 142: *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 142:*Registering Type Factories*

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

The code in Example 142 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 143 a calling `getSupportedNamespaces()` on the generated type factory would return an array of strings containing a single entry: `http://packageTracking.com/packageTypes`.

Example 143: *WSDL Fragment*

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

Example 143:*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 144 shows code calling `getSupportedNamespaces()`.

Example 144:*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 143 on page 207 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 145, would return a QName whose local part is `packageInfo` and whose namespace URI is `http://packageTracking.com/packageTypes`.

Example 145: *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 146 gets an instance of the generated `PackageInfo` object by passing `getJavaType()` the QName of the `packageInfo` XMLSchema type defined in Example 143 on page 207.

Example 146: *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 146 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 8

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

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 204.

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 216.

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 11. These methods automatically set the data type identifier to the appropriate schema type when they store the data.

Table 11: *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 11: *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 147 shows code for storing a `widgetSize` in an `anyType`.

Example 147:*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 148 shows code for storing a `widgetSize` in an `anyType` without providing its `QName`.

Example 148:*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 149 gets the schema type of an anyType and prints it out to the console.

Example 149: *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 12 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 12: *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 150 shows an example of retrieving a widgetSize from an anyType.

Example 150: *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 151 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 151: *Working with anyTypes*

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

AnyType anyOrder;

1

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

2

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

Example 151:*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 151 on page 218 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 9

Using 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

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

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 224

- *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 the JMS transport the properties would include attributes like `messageType` and `destinationStyle`.

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 plug-in, 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, 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 plug-in avoids these drawbacks. The WSDL publishing plug-in 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 plug-in 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` element 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` element, 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 42.
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 152 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 152:*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 36.

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 153 shows the signature for `createReference()`.

Example 153: *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 154 shows the code for generating a reference for a static instance of the `Cling` service.

Example 154: *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.

Getting a bus

Typically, you will receive a reference inside of a service's implementation object and will not have access to the bus which is hosting the current servant. In order to get a handle for a servant's default bus you would use code similar to that shown in Example 155.

Example 155: *Getting a Bus Reference Inside a Servant*

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

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 156.

Example 156: *Bus.createClient()*

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

Example

Example 157 shows the code for creating a service proxy for the Cling service from a reference.

Example 157:*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

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 158 shows a WSDL fragment that imports the reference schema.

Example 158: *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 159.

Example 159: *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 160 shows the definition for a message that contains a reference.

Example 160: *Message with a Reference*

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

Bank interface

The portType element defining the Bank interface defines a single operation named get_account. This operation takes a string as input and returns a reference. Example 161 shows the portType element for the Bank interface.

Example 161: *Bank portType Element*

```
<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 162 shows the complete contract used for the code generated in the following discussions about the factory pattern.

Example 162: *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 162: *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 162:*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 163 shows the generated `BamkImpl` with `accounts` added.

Example 163: *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 163: *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 164 shows the fully implemented `get_account()`.

Example 164: *get_account()*

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

Example 164:*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 164 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 224, the WSDL pointer in a reference can either point to the server process' local copy of the service contract or, if the WSDL publishing plug-in 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 plug-in.

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 165 shows the code for creating an Account service proxy from a reference.

Example 165: *Creating an Account Service Proxy*

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

Code for the client main()

Example 166 shows the completed code for the bank client's main line.

Example 166: *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 166:*Code for Bank Client*

```
    Reference acctRef = bankProxy.get_account(account_name);

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

  // Invoke operations on acctProxy
    }
}
```

The code in Example 166 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

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 158 shows a WSDL fragment that imports the reference schema.

Example 167: *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 159.

Example 168: *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 160 shows the definition for a message that contains a reference.

Example 169: *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` element or `output` element the interface definition includes a dummy input message.

Example 170 shows the `portType` element defining the callback object's interface.

Example 170: *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 171 shows the `portType` element defining the server's interface.

Example 171: *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 172 shows the complete contract used for the code generated in the following discussions about callbacks.

Example 172:*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 172:*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 172:*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 173 shows the code for the callback object.

Example 173: *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 173:*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 174 shows the code for client `main()`.

Example 174: *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 174:*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 174:*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 174 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 18.

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 175 shows the completed `RegisterImpl` class. The code in bold is added to the generated class by the user.

Example 175: *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 175: *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 10

Using Native XML

The Artix Java API provides a utility class that populates Artix generated objects from an XML document. This utility class will also convert Artix generated object back into a native XML representation.

In this chapter

This chapter discusses the following topics:

Populating Artix Objects with XML

Overview

You may have instances where the data your application is using input that is already in XML. For example, your data may stored in a database that stores information as XML or you are working with a word processing document stored in the Oasis Open Document format. The problem them becomes how to populate the objects used in your application with the XML data.

Artix solves this problem by providing the utility class `com.iona.jbus.utils.XMLUtils`. This class provides the overloaded static method `fromXML()` for populating objects using XML data. It uses the XMLSchema definitions of the data the objects store to parse the XML data and populate the elements in the object.

Populating an object generated from an XMLSchema type

If the object you are populating is generated to represent an XMLSchema type, you can use the simple form of `fromXML()`. The signature for this form is shown in Example 176.

Example 176: *fromXML() for Types*

```
static Object fromXML(String xml, QName name,
                      Class class, String path)
```

`fromXML()` returns an `Object` that can be cast into the appropriate type. It takes four arguments:

`String xml`	Contains the XML data to populate the object.
`QName name`	Specifies the QName of the XMLSchema type from which the object was generated.
`Class class`	Specifies the `Class` object for the object to be populated.
`String path`	Specifies the path to the contract or XMLSchema document defining the data the object represents.

If, for example, your application works with student records whose structure is defined as an XMLSchema complex type called `studentRec`, and it reads records from an XML database, the code for populating the object would be similar to that shown in Example 177.

Example 177: *Populating an Object from XML*

```
1  FileInputStream file = new FileInputStream("test.xml");

2  byte record[256];
   file.read(record);

3  String xmlRec = new String(record);

4  QName name = new QName("schemas.com/tests/types",
                          "studentRec");

5  StudentRec student = (StudentRec)XMLUtil.fromXML(xmlRec, name,
                                                    StudentRec.class,
                                                    "./grader.wsdl");
```

The code in Example 177 does the following:

1. Opens a file containing XML data
2. Reads in a record from the file.
3. Converts the byte stream into a `String`.
4. Creates the QName for the type definition.
5. Uses the `XMLUtils` class to populate a `StudentRec` object with the XML data read from the file.

If the XML data passed into `fromXML()` does not conform to the XMLSchema definition for the type a `WriteException` will be thrown.

Populating an object generated from an XMLSchema element

If the object you are populating is generated to represent an XMLSchema element, you can use the more flexible form of `fromXML()`. This form will work with both XMLSchema types and XMLSchema elements. The signature for this form is shown in Example 178.

Example 178: *Five Argument form of fromXML()*

```
static Object fromXML(String xml, QName elementName,
                      QName typeName, Class class, String path)
```

`fromXML()` returns an `Object` that can be cast into the appropriate type. It takes five arguments:

`String xml`	Contains the XML data to populate the object.
`QName elementName`	Specifies the QName of the XMLSchema element from which the object was generated.
`QName typeName`	Specifies the QName of the XMLSchema type from which the object was generated.
`Class class`	Specifies the `Class` object for the object to be populated.
`String path`	Specifies the path to the contract or XMLSchema document defining the data the object represents.

If your object represents an XMLSchema element, you would specify `null` for `typeName`. Conversely, if your object represents an XMLSchema type, you would specify `null` for `elementName`.

If we changed Example 177 so that `studentRec` was defined as an XMLSchema element instead of a complex type, the code for populating the object would be similar to that shown in Example 179.

Example 179:*Populating an Object from XML*

```
FileInputStream file = new FileInputStream("test.xml");

byte record[256];
file.read(record);

String xmlRec = new String(record);

QName name = new QName("schemas.com/tests/types",
                       "studentRec");

StudentRec student = (StudentRec)XMLUtil.fromXML(xmlRec, name,
                                                 null
                                                StudentRec.class,
                                                "./grader.wsdl");
```

The code in Example 179 differs from the code in Example 177 in only one way. The call to `fromXML()` includes the extra parameter. In this case, because `studentRec` is defined as an element it is `null`.

If the XML data passed into `fromXML()` does not conform to the XMLSchema definition for the element a `WriteException` will be thrown.

Converting Artix Objects Into XML

Overview

All Artix generated objects have a `toString()` method that will produce a stringified representation of the object. There are instances that you need to recreate the XML data represented by the object. For example, you may need to store the data in an XML database. Recreating the XML data represented by an object can also be a useful debugging tool.

Artix solves this problem by providing the utility class `com.iona.jbus.utils.XMLUtils`. This class provides the overloaded static method `toXML()` for converting objects into their XML form. It uses the XMLSchema definitions of the XML data the objects represent. From the XMLSchema definition, Artix can decompile the Java object and parse it into valid XML data.

Artix objects that represent an XMLSchema type

If the object you are converting into XML was generated by Artix to represent an XMLSchema type you can use the simplest form of `toXML()`. The signature for this form is shown in Example 180.

Example 180: *Two Argument toXML()*

```
static String toXML(Object obj, String path)
```

It returns a `String` containing the XML representation of the object and takes two arguments.

`Object obj`	Specifies the object you are converting to XML. This object must have been generated by the Artix Java code generator because it uses Artix specific code for determining the QName of the type which the object represents.
`String path`	Specifies the path to the contract or XMLSchema document defining the data the object represents.

Objects that represent an XMLSchema type

If you have an object, that was not generated by Artix, that represents an XMLSchema type and you have access to the XMLSchema document that defines the type, you can still convert it into XML. `toXML()` has a three

argument form that allows you to specify the QName of the XMLSchema type the object represents. The signature for this form is shown in Example 181.

Example 181:*Three Argument toXML()*

```
static String toXML(QName name, Object obj, String path)
```

It returns a `String` containing the XML representation of the object and takes three arguments.

`QName name`	Specifies the QName of the XMLSchema type represented by the object.
`Object obj`	Specifies the object you are converting to XML. This object must have been generated by the Artix Java code generator because it uses Artix specific code for determining the QName of the type which the object represents.
`String path`	Specifies the path to the contract or XMLSchema document defining the data the object represents.

Objects that represent an XMLSchema element

If you have an object, that represents an XMLSchema element and you have access to the XMLSchema document that defines the type, can convert it into XML using the four argument form of `toXML()`.This form that allows you to specify the QName of the XMLSchema element the object represents. It also allows you to convert an object that represents an XMLSchema type by specifying the type's QName. The signature for this form is shown in Example 182.

Example 182:*Four Argument toXML()*

```
static String toXML(QName elementName, QName typeName,
                    Object obj, String pth)
```

It returns a `String` containing the XML representation of the object and takes four arguments.

`QName elementName`	Specifies the QName of the XMLSchema element represented by the object.
`QName typeName`	Specifies the QName of the XMLSchema type represented by the object.

`Object obj`	Specifies the object you are converting to XML. This object must have been generated by the Artix Java code generator because it uses Artix specific code for determining the QName of the type which the object represents.
`String path`	Specifies the path to the contract or XMLSchema document defining the data the object represents.

If your object represents an XMLSchema element, you would specify `null` for `typeName`. Conversely, if your object represents an XMLSchema type, you would specify `null` for `elementName`.

Converting References into XML

Overview

Artix references are defined with in an Artix specific XMLSchema document that is not always available to applications. Therefore, they contain enough information to be self-describing. For converting them to and from XML, `XMLUtils` provides special methods.

Converting to XML

To convert an Artix reference into XML, you use `XMLUtils.referenceToXML()`. `referenceToXML()` takes a single `Reference` object and returns a `String` object containing the XML representation of the reference. If it cannot convert the reference it throws a `WriteException`.

Converting from XML

To convert the XML representation of an Artix reference into an Artix `Reference` object, you use `XMLUtil.referenceFromXML()`. `referenceFromXML()` takes a `String` object containing the XML representation of the reference and returns the `Reference` object constructed from the XML. If the supplied XML is not valid a `ReadException` is thrown.

CHAPTER 11

Using Message Contexts

Artix implements and extends the JAX-RPC 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

Overview

Artix implements the JAX-RPC `MessageContext` interface. JAX-RPC message contexts are primarily used in writing 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. This additional information can include SOAP headers, GIOP context objects, transport attributes, and MIME type definitions.

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 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 5.

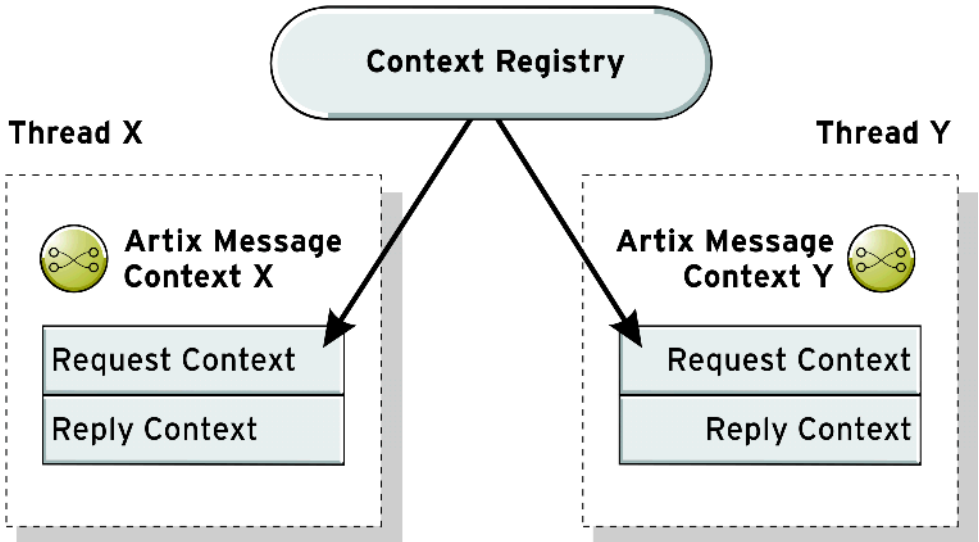

Figure 5: *Overview of the Message Context Mechanism*

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 204.
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 JAX-RPC message context, a SOAP message context, or an Artix message context. Both the JAX-RPC interface and the Artix message context interface 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 JAX-RPC message context interface, so all of the JAX-RPC message context methods are available after you cast a JAX-RPC message context to an Artix message context.

The SOAP message context, which is defined by the JAX-RPC specification, is only available when using the Artix SOAP binding. It provides access to messages in SOAP form. Using this context you can manipulate the messages using the `SOAPMessage` APIs. For more information see, "Working with SOAP Messages" on page 513.

In this section

This section discusses the following topics:

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 183 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 183: *getContextRegistry()*

```
ContextRegistry com.iona.jbus.Bus.getContextRegistry();
```

To get access to the context registry from your application code, do the following:

1. Get a handle for the desired bus using one of the following methods as shown in "Getting a Bus" on page 42.
2. Call `getContextRegistry()` on the returned bus to get a reference to the context registry.

Example

Example 184 shows an example of getting the context registry from within the implementation object of an Artix service.

Example 184: *Getting the Context Registry*

```
import java.net.*;
import java.rmi.*;
1  import com.iona.jbus.*;

public class Atherny
{
// get the bus

2  ContextRegistry contReg = bus.getContextRegistry();
```

Example 184:*Getting the Context Registry*

```
...
}
```

The code in Example 184 does the following:

1. Import the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. 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 JAX-RPC `MessageContext` objects. These objects provide access to properties stored in the contexts using the APIs defined in the JAX-RPC specification.

Artix provides two means of getting the current 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 JAX-RPC `MessageContext` objects. Example 185 shows the signature for `getCurrent()`.

Example 185: *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 282.

Example 186 shows how to get an message context from the context registry.

Example 186:*Getting a Message Context*

```
import java.net.*;
import java.rmi.*;
import javax.xml.rpc.handlers.*;

1  import com.iona.jbus.*;

public class Atherny
{
// get the bus

2  ContextRegistry contReg = def_bus.getContextRegistry();

3  MessageContext messCont = contReg.getCurrent();
...
}
```

The code in Example 186 does the following:

1. Import the package `com.iona.jbus` so that it has access to the Artix bus APIs.
2. Call `getContextResistry()` on the default bus to get the default bus' context registry.
3. 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 JAX-RPC `MessageContext` objects. Example 187 shows the signature for `getCurrentMessageContext()`.

Example 187:*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 282.

Example 188 shows how to get an message context using the `DispatchLocals` interface.

Example 188: *Getting a Message Context*

```
import java.net.*;
import java.rmi.*;
import javax.xml.rpc.*

import com.iona.jbus.*;

public class Atherny
{
  MessageContext messCont =
    DispatchLocals.getCurrentMessageContext();
  ...
}
```

Working with JAX-RPC 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 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 handlers are only available to 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.

JAX-RPC 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.

Artix context properties

Artix has a number of standard properties that it stores in the JAX-RPC message context. These properties can all be accessed using the appropriate constant from the `com.iona.jbus.ContextConstants` class. Table 13 lists the context properties used by Artix.

Table 13: *Artix Context Properties*

Property	Description
`OPERATION_NAME`	Holds the name of the operation the originated the message being processed. See "Working with Operation Parameters" on page 508.
`SERVER_REQUEST_CLASSES`	Holds an array of `Class` objects representing the classes of each part of the current request message. See "Working with Operation Parameters" on page 508.

Table 13: *Artix Context Properties*

Property	Description
SERVER_REQUEST_VALUES	Holds an array of `Object` objects containing the data for each part in the current request message. See "Working with Operation Parameters" on page 508.
SERVER_RESPONSE_CLASSES	Holds an array of `Class` objects representing the classes of each part of the current response message. See "Working with Operation Parameters" on page 508.
SERVER_RESPONSE_VALUES	Holds an array of `Object` objects containing the data for each part in the current response message. See "Working with Operation Parameters" on page 508.
CLIENT_REQUEST_CLASSES	Holds an array of `Class` objects representing the classes of each part of the current request message. See "Working with Operation Parameters" on page 508.
CLIENT_REQUEST_VALUES	Holds an array of `Object` objects containing the data for each part in the current request message. See "Working with Operation Parameters" on page 508.
CLIENT_RESPONSE_CLASSES	Holds an array of `Class` objects representing the classes of each part of the current response message. See "Working with Operation Parameters" on page 508.
CLIENT_RESPONSE_VALUES	Holds an array of `Object` objects containing the data for each part in the current response message. See "Working with Operation Parameters" on page 508.

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 189 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 189:*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 handlers are only available to handlers further down the handler chain and expire once the operation's invocation is completed. For more information about handlers, see "Writing Handlers" on page 479.

Example 190 shows the code for setting a property in the request context.

Example 190:*Setting a Property in a Message Context*

```
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
// get the bus

2 ContextRegistry contReg = def_bus.getContextRegistry();

3 MessageContext context = contReg.getCurrent();

4 boolean isEncrytped = TRUE;

5 context.setProperty("isEncrypted", isEncrypted);

...
}
```

The code in Example 190 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
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. Creates the an instance of the property's class and set the values.
5. 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 191 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 191: *MessageContext.getProperty()*

```
Object getProperty(String name);
```

Example 192 shows the code for getting a SOAP header from the request context.

Example 192: *Getting a Property from the Message Context*

```
import java.net.*;
import java.rmi.*;
1  import com.iona.jbus.*;

public class Atherny
{
// get the bus

2  ContextRegistry contReg = def_bus.getContextRegistry();

3  MessageContext context = contReg.getCurrent();

4  boolean encrypt = (boolean)context.getProperty("isEncrypted");
...
}
```

The code in Example 192 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
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. 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 193 shows the signature for `removeProperty()`. It takes a single parameter, `name`, that represents the name of the property you wish to remove.

Example 193: *MessageContext.removeProperty()*

```
void removeProperty(String name);
```

Example 194 shows the code for removing a property from the message context.

Example 194: *Removing a Property from a Message Context*

```
import java.net.*;
import java.rmi.*;
import com.iona.jbus.*;

public class Atherny
{
// get the bus
1 ContextRegistry contReg = def_bus.getContextRegistry();

2 MessageContext context = contReg.getCurrent();

3 context.removeProperty("isEnctryted");
...
}
```

The code in Example 194 does the following:

1. Calls `getContextResistry()` on the default bus to get the default bus' context registry.
2. Calls `getCurrent()` on the context registry to get the message context for the application's thread.
3. 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 195 shows the code for seeing if a property is set in the message context.

Example 195: *Querying a Property in the Message Context*

```
import java.net.*;
import java.rmi.*;
import com.iona.jbus.*;

public class Atherny
{
// get the bus
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 6 shows how context properties are passed with messages.

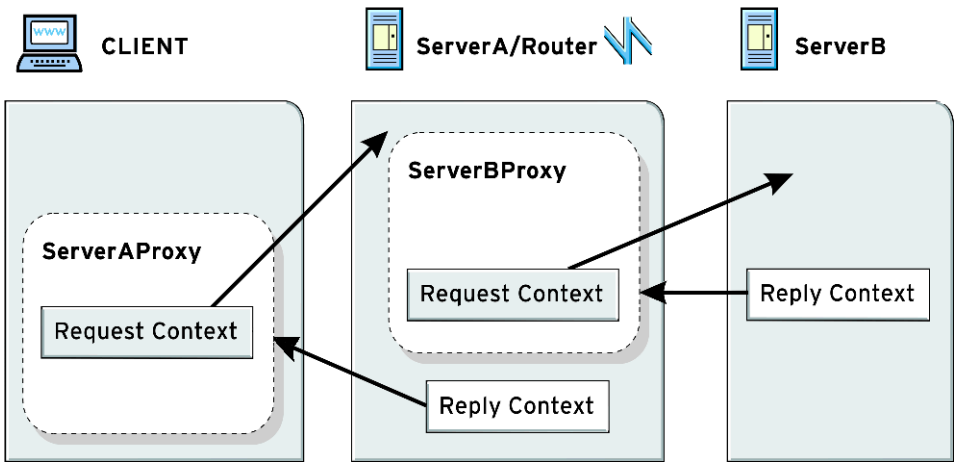

Figure 6: *Contexts Passed Along Request/Reply Chain*

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 two setter methods: `setReplyContext()` and `setRequestContext()`. Example 196 shows the signature for these methods.

Example 196: *Methods for Setting a Property*

```
void setReplyContext(QName name, Object value);
void setRequestContext(QName name, Object 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 197 shows the code for setting a property in the request context.

Example 197:*Setting a Property in an Artix Message Context*

```
import java.net.*;
import java.rmi.*;
1 import com.iona.jbus.*;

public class Atherny
{
// get the bus
2 ContextRegistry contReg = def_bus.getContextRegistry();

3 IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4 MusicTagType tag = new MusicTagType();
tag.setArtist("Murphy");
tag.setAlbum("Law");

5 QName contextName = new QName("http://records.com/",
                                "MusicTags");

6 context.setRequestContext(contextName, tag);

...
}
```

The code in Example 197 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
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 and casts it to an Artix message context.
4. Creates the an instance of the property's class and set the values.
5. Creates the QName for the property.
6. Sets the property by calling `setRequestContext()` with the appropriate `QName` and the newly created property object.

Getting a property

Artix message contexts have two methods that allows you to get a property from one of the context containers. These methods are `getReplyContext()` and `getRequestContext()`. Example 198 shows the signature for these methods.

Example 198: *Methods for Getting a Property*

```
Object getReplyContext(QName name);
Object getRequestContext(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. Artix has a number of preregistered context types to support transport attributes. In addition, You can register your own properties to use as SOAP headers or GIOP service contexts.

Example 199 shows the code for getting a property from the request context.

Example 199: *Getting a Property*

```
   import java.net.*;
   import java.rmi.*;
1  import com.iona.jbus.*;

   public class Atherny
   {
   // get the bus
2  ContextRegistry contReg = def_bus.getContextRegistry();

3  IonaMessageContext context =
     (IonaMessageContext)contReg.getCurrent();

4  QName refName = new QName("http://records.com/","MusicTags");
5  MusicTagType tag =
     (MusicTagType)context.getRequestContext(refName);
   ...
   }
```

The code in Example 199 does the following:

1. Imports the package `com.iona.jbus` so that it has access to the Artix bus APIs.
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 and casts it to an Artix message context.
4. 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.
5. 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 199 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.

Special Properties

Artix message contexts have two special properties for use by servers:

- `oneway` is a `boolean` property that specifies if a request requires a response.
- `correalationID` is stored as a `long` and specifies a unique identifier that allows a server to correlate an incoming request with its corresponding outgoing reply.

The `oneway` property is available in a server's Artix message context once a message reaches the request-level interceptors. You can check its value using `IonaMessageContext.isOneway()`. If the request is a oneway request, meaning that it will not generate a reply, `oneway` is `true`. For requests that require a response, `oneway` is `false`.

Example 200 shows code for checking if a request is oneway.

Example 200:*Seeing if a Request is Oneway*

```
1  import com.iona.jbus.IonaMessageContext;

   ...
2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  if (context.isOneway())
   {
     System.out.println("This is a oneway request.");
   }
```

Example 200 does the following:

1. Imports the proper `jbus` package.
2. Gets the context registry.
3. Gets the Artix message context.
4. Determine if the request is oneway.

The `correlationID` property is available at all levels of the server-side messaging chain and is accessed using `IonaMessageContext.getCorrelationID()`. The value of the property is a `long` that is specific to each request/reply pair. Using `correlationID` you could, for instance, write an interceptor that tracked the amount of time required for a reply to be generated for each request.

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. How you define the data for the context and how you register the context will depend on the payload format the application uses.

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 204.
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

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 201.

Example 201: *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 202.

Example 202: *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 201.

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.

In this section

This section discusses the following topics:

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 203 shows the signature of the `registerContext()` function used to register a context to be used as a SOAP header.

Example 203: *The registerContext() Function for SOAP Headers*

```
void ContextRegistry.registerContext(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 202 on page 290 you would use code similar to Example 204.

Example 204: *Registering a SOAP Header Property*

```
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 204 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.

Registering a context to use as a CORBA Header

To register a property to be used as a CORBA header you need to provide an ID to be placed in the GIOP service context ID.

Example 205 shows the signature of the `registerContext()` function used to register a property to be used as a CORBA header.

Example 205: *The registerContext() Function for CORBA Headers*

```
void ContextRegistry.regiserContext(QName name, QName type,
                                    long context_id);
```

This `registerContext()` method takes the following arguments:

`name`	The qualified name used to represent the property.
`type`	The qualified name of the property's data type.
`context_id`	The ID that tags the GIOP service context containing the Artix context. In CORBA, the `context_id` corresponds to a service context ID of `IIOP::ServiceId` type. For details of GIOP service contexts, consult the OMG CORBA specification.

For example, to register a CORBA header property of the type defined in Example 202 on page 290 you would use code similar to Example 206.

Example 206: *Registering a Property as a CORBA Header*

```
1  // Artix servant, servant, obtained earlier
   headerInfoTypeFactory fact = new headerInfoTypeFactory();
   servant.registerTypeFactory(factArray);
```

Example 206:*Registering a Property as a CORBA Header*

```
2   // Bus, bus, obtained earlier
    ContextRegestry contReg = bus.getContextRegistry();

3   // Create a QName for the new property
    QName name = new QName("http://javaExamples.iona.com",
                           "CORBAHeader");

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   // Register the property
    contReg.registerContext(name, type, 1);
```

The code in Example 206 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` 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. Register the property with the context registry.

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 in 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 207 on page 296 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 202 on page 290. 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 207:*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"/>
      <element name="SOAPHeaderInfo" type="ns1:headerInfo"/>
    </schema>
  </types>
  <message name="greetMeRequest">
    <part element="requestType" name="me"/>
  </message>
  <message name="greetMeResponse">
    <part element="responseType" name="theResponse"/>
  </message>
  <message name="headerMsg">
    <part element="SOAPHeaderInfo" name="headerPart" />
  </message>
  <portType name="Greeter">
    <operation name="greetMe">
      <input message="greetMeRequest" name="greetMeRequest"/>
      <output message="greetMeResponse" name="greetMeResponse"/>
    </operation>
  </portType>
```

Example 207: *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 208 on page 298 shows the code for the client.

Example 208: *Client Code*

```
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();

5   QName name = new QName("http://javaExamples.iona.com",
                           "SOAPHeader");
```

Example 208:*Client Code*

```
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 208 on page 298 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 209 on page 300 shows the code for the server's main line.

Example 209: *Server main()*

```
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 209: *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 209 on page 300 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 210 shows the code for the `GreeterImpl` implementation object.

Example 210: *Implementation of the Greeter Service*

```
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 def_bus = DispatchLocals.getCurrentBus();

2   ContextRegestry contReg = bus.getContextRegistry();
```

Example 210:*Implementation of the Greeter Service*

```
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 210 on page 302 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 12

Working with Transport Attributes

Using the Artix context mechanism, you can set many of the the transport attributes at runtime.

In this chapter

This chapter discusses the following topics:

How Artix Stores Transport Attributes

Overview

Artix uses the context mechanism described in "Using Message Contexts" on page 267 to store the properties used to configure the transport layer and populate any headers used by the selected transport. Most of the properties are stored in the normal Artix context containers. However, some properties that are used in initializing the transport layer at start-up are stored in a special context container.

Initialization properties

Some transport attributes, such as JMS broker sign-on values or a server's HTTP endpoint URL, are used by Artix when it is initializing the transport layer. Therefore, they need to be specified before Artix it initializes the transport layer for a service or a service proxy. These attributes are stored in a special context container. When the bus initializes the transport layer, it will check this special context container for any initialization properties.

Global transport attributes

For most transport properties such as HTTP keep-alive, WebSphere MQ `AccessMode`, and Tib/RV `callbackLevel`, the context objects containing the transport's properties are stored in the Artix request context container and the Artix reply context container. Once you have retrieved the context object from the proper context container, you can inspect the values of transport headers and other transport related properties such as codeset conversion. You can also dynamically set many of the values for outgoing messages using the context APIs. For a full listing of all the possible port attributes for each transport see Designing Artix Solutions.

Transport specific

Transport attributes are stored in built-in contexts. These contexts are preregistered with the context container when the transport layer is initialized. They are specific to the different transports. For example, if you request the context for the HTTP port attributes from the context container, the returned context will have methods for setting and examining HTTP specific attributes. However, if the application is using another transport, WebSphere MQ for example, the HTTP configuration context will not be registered and you will be unable to get the HTTP configuration context from the container.

When are the attribute contexts populated

All of the transport attributes have default values that are specified in either the service's contract or in the service's configuration. If you do not use the contexts for overriding transport attributes, these are always used when sending messages. However, when you get the transport attributes for an outgoing message, the context will be empty. You will need to create an instance of the context and set the values you want to override in the context yourself.

When a message is received by the transport layer, the transport populates the context with the attributes of the message it receives. For example, if you are using HTTP the values of the incoming message's HTTP header will be used to populate the context. The context can then be inspected at any point in the application's code.

Getting Transport Attributes from an Artix Context

Overview

All of the contexts for holding transport attributes are handled using the standard context mechanism. To get a transport attribute context do the following:

1. Get the applications message context registry as shown in "Getting the Context Registry" on page 271.
2. Get the message context for the current application as shown in "Getting the Message Context for a Thread" on page 273.
3. Cast the message context to an Artix message context.
4. Get the desired context from the appropriate context container.

Once you have the context you can inspect it and set new values for any of its properties.

Getting a transport attribute context

You get an instance of a transport attribute context from an Artix message context using the standard context APIs discussed in "Working with Artix Message Contexts" on page 282. To make it easy to remember the names used to access each context, Artix provides a helper class called `ContextConstants` that has a static member for each configuration context. The static member name for each configuration context is shown in Table 14.

Table 14: *Configuration Context QNames*

Context	ContextConstants Member
HTTP Client Incoming Attributes	`HTTP_CLIENT_INCOMING_CONTEXTS`
HTTP Client Outgoing Attributes	`HTTP_CLIENT_OUTGOING_CONTEXTS`
HTTP Server Incoming Attributes	`HTTP_SERVER_INCOMING_CONTEXTS`
HTTP Server Outgoing Attributes	`HTTP_SERVER_OUTGOING_CONTEXTS`
CORBA Transport Attributes	`CORBA_CONTEXT_ATTRIBUTES`

Table 14: *Configuration Context QNames*

Context	ContextConstants Member
MQ Connection Attributes	`MQ_CONNECTION_ATTRIBUTES`
MQ Outgoing Message Attributes	`MQ_OUTGOING_MESSAGE_ATTRIBUTES`
MQ Incoming Message Attributes	`MQ_INCOMING_MESSAGE_ATTRIBUTES`
JMS Client Header Attributes	`JMS_CLIENT_CONTEXT`
JMS Server Header Attributes	`JMS_SERVER_CONTEXT`
i18n Server Attributes	`I18N_INTERCEPTOR_SERVER_QNAME`
i18n Client Attributes	`I18N_INTERCEPTOR_CLIENT_QNAME`
Bus Security Attributes	`SECURITY_SERVER_CONTEXT`

Once you have gotten the desired context from the Artix message context, you will need to cast it to the appropriate class for the context. Table 15 lists the data types for each of the configuration contexts.

Table 15: *Configuration Context Classes*

Context	Class
HTTP Client Attributes	`com.iona.schemas.transports.http.configuration.context.ClientType`
HTTP ServerAttributes	`com.iona.schemas.transports.http.configuration.context.ServerType`
CORBA Attributes	`com.iona.schemas.bindings.corba.contexts.CORBAAttributesType`
MQ Connection Attributes	`com.iona.schemas.transports.mq.context.MQConnectionAttributesType`
MQ Message Attributes	`com.iona.schemas.transports.mq.context.MQMessageAttrinutesType`
JMS Client Header Attributes	`com.iona.schemas.transports.jms.context.JMSClientHeadersType`
JMS Server Header Attributes	`com.iona.schemas.transports.jms.context.JMSServerHeadersType`
i18n Server Attributes	`com.iona.schemas.bus.i18n.context.ServerConfiguration`
i18n Client Attributes	`com.iona.schemas.bus.i18n.context.ClientConfiguration`
Bus Security Attributes	`com.iona.schemas.bus.security_context.BusSecurity`

Setting Configuration Attributes

Overview

Depending on the attributes that are being set, you will one of two methods for setting the configuration information into the context container. For most cases, you will use the standard context mechanism. For properties that must be known before the bus initializes the transport layer, you will use the specialized configuration context.

In this section

This section discussed the following topics:

Using the Standard Contexts

Durability of settings

When programmatically alter your application's transport attributes, you override any settings read from the application's contract and the application's configuration file. The durability of this setting depends on whether the application is a server or a client.

For servers, transport attribute settings are valid only for a single request. After each request is processed and a reply is sent the settings revert back to the settings specified in the contract.

For clients, the contexts used to programatically set transport attributes are permanent. Once set, a value remains in place until it is explicitly changed. So, if you change a client's HTTP username attribute to `GreenDragon`, it will be used in all future requests. Exceptions to this rule are noted when applicable.

Configuring clients

To override the default transport attributes on the client-side you set values on the context in the request context container. The bus uses the values from the request context container to override the default configuration on the client's transport before sending a request. If no values have been set in the request context container the transport uses its default values.

The values in a client's reply context are set by the Artix bus when a reply is received by the transport layer. They can be checked by client code at any point.

Configuring servers

To override the default transport attributes on the server-side you set the values on the contexts in the reply context container. The bus uses the values from the reply context container to override the default configuration on the server's transport before sending a reply. If no values have been set in the reply context container the transport uses its default values.

The values in a server's request context are set by the Artix bus when a request is received by the transport layer. The properties can be checked at any point in the server's messaging chain and in the server's implementation object.

Using the Configuration Context

Overview

There are a few transport attributes that need to be specified before the transport layer of an Artix application is instantiated. For example when using a secure JMS broker, your application need to know its username and password before it attempts to connect to the JMS broker. To accomplish this, you need to set these properties before the user level code is registered with the bus. Artix uses a special context, called the configuration context, to do this.

Available properties

Currently, Artix supports two special port properties:

- HTTP Endpoint URL - specifies the URL on which the server can be contacted.
- JMS Broker Connection Security Info - specifies the username and password used by an application when connecting to the JMS broker.

Procedure

To register a special port property do the following:

1. Get the configuration context from the context registry.
2. Get a copy of the desired property from the configuration context.
3. Set the appropriate values into the property.
4. If the application is a server, register the servant with the bus.
5. If the application is a client, instantiate the service proxy.

Getting the configuration context

The configuration context is obtained directly from the context registry using the `getConfigurationContext()` method shown in Example 211. It is returned as a port specific `ContextContainer` object. To specify the port with which the context container is associated you pass in the `QName` of the

service defining the port and the name of the port. You can also specify if the bus will create an instance of the configuration context for the specified port.

Example 211:*getConfigurationContext()*

```
ContextContainer getConfigurationContext(QName serviceName,
                                         String portName,
                                        boolean createIfNotFound);
```

Setting properties in the configuration context

Once you have the context container for the configuration context, you can set the desired port properties. Like a normal message context, the context container has a `getContext()` method for retrieving contexts from the container and a `setContext()` method for writing new contexts to the container.

`getContext()`, shown in Example 212, gets the instance of a context from the container. The method can also create a new instance of the desired context. The context is returned as a Java `Object` that can then be cast into the appropriate data type. Once you have the context object, you can manipulate any data set in it and the changes are propagated back to the container.

Example 212:*getContext()*

```
Object getContext(QName contextName, boolean createIfNotFound);
```

You can also use the `setContext()` method, shown in Example 213, to set a context into the context container. `setContext()` takes an instance of the context's data type and the context name. The context instance is then use to populate the context. All of the values set on the context instance become the values used to configure your server port.

Example 213:*setContext()*

```
void setContext(QName contextName, Object context);
```

Setting HTTP Attributes

Overview

Artix uses four contexts to support the HTTP transport. Two contexts support the server-side HTTP information. The server-side contexts are of type `com.iona.schemas.transports.http.configuration.context.ServerType`. The other two contexts support the client-side HTTP information. The client-side contexts are of type `com.iona.schemas.transports.http.configuration.context.ClientType`.

The information stored in the HTTP transport attribute contexts correlates to the values passed in an HTTP header.

In this section

This section discusses the following topics:

Client-side Configuration

Overview

HTTP clients have access to both the values being passed in the HTTP header of the outgoing request and the values received in the HTTP header of the response. The information for each header is stored in a separate context.

Outgoing header information

On the client-side, the outgoing context, `HTTP_CLIENT_OUTGOING_CONTEXT`, is available in the client's request context. Any changes made to values in the outgoing context are placed in the request's HTTP header and propagated to the server. For example, if you want to allow requests to be automatically redirected you could set the `AutoRedirect` attribute to `true` in the client's outgoing context. Example 214 shows the code for setting the `AutoRedirect` property for a client.

Example 214: *Setting a Client's AutoRedirect Property*

```
1  import com.iona.schemas.transports.http.configuration.context.*;
   import com.iona.jbus.ContextConstants;

   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  ClientType httpAtribs =
      (ClientType)context.getRequestContext(ContextConstants.HTTP_C
      LIENT_OUTGOING_CONTEXT, true);

5  httpAtribs.setAutoRedirect(true);

   // make proxy invocations
```

The code in Example 214 does the following:

1. Imports the package containing the HTTP client context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.

4. Gets the client's outgoing HTTP context from the request context container.
5. Sets the value of the `AutoRedirect` property to `true`.

Outgoing client attributes

Table 16 shows the attributes that are valid in the outgoing HTTP client context.

Table 16: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
`Accept`	`String getAccept()` `void setAccept(String val)`	Specifies the MIME types the client can handle in a response.
`Accept-Encoding`	`String getAcceptEncoding()` `void setAcceptEncoding(String val)`	Specifies the types of content encoding the client can handle in a response. This property typically refers to compression mechanisms.
`Accept-Language`	`String getAcceptLanguage()` `void setAcceptLanguage(String val)`	Specifies the language the client prefers. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, `en-US`.
`Authorization`	`String getAuthorization()` `void setAuthorization(String val)`	Specifies the credentials that will be used by the server to authorize requests from the client.

`AuthorizationType`	`String getAuthorizationType()`
	`void setAuthorizationType(`
	`String val)`
		Specifies the name of the authentication scheme in use.

`AutoRedirect`	`Boolean isAutoRedirect()` `void setAutoRedirect(Boolean val)`	Specifies whether a request should be automatically redirected by the server. The default is `false` to specify that requests are not to be automatically redirected.

Table 16: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
`BrowserType`	`String getBrowserType()` `void setBrowserType(String val)`	Specifies information about the browser from which the request originates. This property is also know as the user-agent.
`Cache-Control`	`String getCacheControl()` `void setCacheControl(String val)`	Specifies directives to caches along the request/response path. Valid values are: **`no-cache`**: caches must revalidate responses with the server. If response header fields are given, the restriction applies only to those header fields. **`no-store`**: caches must not store any part of a request or its response. **`max-age`**: the max age, in seconds, of an acceptible response. **`max-stale`**: the client will accept expired messages. If a value is given, it specifies the how many seconds after a response expires that the it is still acceptable. If no value is given, all stale responses are acceptable. **`min-fresh`**: the response must stay fresh for the given number of seconds. **`no-transform`**: caches must not modify the media type or the content location of a response. **`only-if-cached`**: caches should return only cached responses.

Table 16: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`ClientCertificate`	`String getClientCertificate()`
	`void setClientCertificate(`
	`String val)`
		Specifies the full path to the PKCS12-encoded X509 certificate issued by the certificate authority for the client.

`ClientCertificateChain`	`String getClientCertificateChain()`
	`void setClientCertificateChain(`
	`String val)`
		Specifies the full path to the file containing all of the certificates in the chain.

`ClientPrivateKey`	`String getClientPrivateKey()`
	`void setClientPrivateKey(`
	`String val)`
		Specifies the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by `ClientCertificate`.

`ClientPrivateKeyPassword`	`String getClientPrivateKeyPassword()`
	`void setClientPrivateKeyPassword(`
	`String val)`
		Specifies the password used to decrypt the PKCS12-encoded private key.

`Connection`	`String getConnection()` `void setConnection(String val)`	Specifies whether a connection is to be kept open after each request/response transaction. Valid values are: **`close`**: the connection is closed after each transaction. **`Keep-Alive`**: the client would like the conneciton to remain open. Servers do not have to honor this request.
`Cookie`	`String getCookie()` `void setCookie(String val)`	Specifies a static cookie that is sent along with a request. **Note:** According to the HTTP 1.1 specification, HTTP cookies must contain US-ASCII characters.
`Expires`	`String getExpires()` `void setExpires(String val)`	Specifies the date after which responses are considered stale.

Table 16: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
`Host`	`String getHost()` `void setHost(String val)`	Specifies the Internet host and port number of the service for which the request is targeted.
`Password`	`String getPassword()` `void setPassword(String val)`	Specifies the password to use in username/password authentication.
`Pragma`	`String getPragma()` `void setPragma(String val)`	Specifies implementation-specific directives that might apply to any recipient along the request/response chain.

`Proxy-Authorization`	`String getProxyAuthroization()`
	`void setProxyAuthentication(`
	`String val)`
		Specifies the credentials used to perform validation at a proxy server along the request/response chain. If the proxy uses username/password validation, this value is not used.

`ProxyAuthorizationType`	`String getProxyAuthorizationType()`
	`void setProxyAuthorizationType(`
	`String val)`
		Specifies the type of authentication used by proxy servers along the request/response chain.

`ProxyPassword`	`String getProxyPassword()` `void setProxyPassword(String val)`	Specifies the password used by proxy servers for authentication if username/password authentication is in use.
`ProxyServer`	`String getProxyServer()` `void setProxyServer(String val)`	Specifies the URL of the proxy server, if one exists, along the request/response chain. **Note:** Artix does not support the existence of more than one proxy server along the request/response chain.

Table 16: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
`ProxyUserName`	`String getProxyUsername()` `void setProxyUserName(String val)`	Specifies the username used by proxy servers for authentication if username/password authentication is in use.
`RecieveTimeout`	`Integer getRecieveTimeout()` `void setRecieveTimeout(Integer val)`	Specifies the number of milliseconds the client will wait to receive a response from a server before timing out. The default is `3000`.
`Referer`	`String getReferer()` `void setReferer(String val)`	Specifies the entity that referred the client to the target server.
`Send-Timeout`	`Integer getSendTimeout()` `void setSendTimeout(Integer val)`	Specifies the number of milliseconds the client will continue trying to send a request to the server before timing out.
`ServerDate`	`String getServerDate()` `void setServerDate(String val)`	Specifies the time setting for the server. When this value is set, the client will use it as the base time from which to calculate message expiration. The client defaults to using its internal system clock.

`Trusted Root Certificate`	`String getTrustedRootCertificates()`
	`void setTrustedRootCertificates(`
	`String val)`
		Specifies the full path to the PKCS12-encoded X509 certificate for the certificate authority.

`Username`	`String getUserName()` `void setUserName(String val)`	Specifies the username used for authentication when the server uses username/password authentication.

Table 16: *Outgoing HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description

`Use Secure Sockets`	`Boolean isUseSecureSockets()`
	`void setUseSecureSockets(`
	`Boolean val)`
		Specifies the client wants to use a secure connection. Secure HTTP connections are also referred to as HTTPS.
		Valid values are `true` and `false`.
		Note: If the contract specifies HTTPS, this value is always `true`.

Incoming header

The client's incoming context, `HTTP_CLIENT_INCOMING_CONTEXT`, is available in the client's reply context after a response from the server has been received by the transport layer. The values stored in this context are for informational purposes only. For example, if you need to check the MIME type of the data returned in the request, you would read it from the client's incoming context as shown in Example 215.

Example 215:*Reading the Content Type in an HTTP Client*

```
1  import com.iona.schemas.transports.http.configuration.context.*;
   import com.iona.jbus.ContextConstants;

   ...
2  // make proxy invocation
   ...

3  ContextRegistry contReg = bus.getContextRegistry();
4  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

5  ClientType httpAtribs =
      (ClientType)context.getReplyContext(ContextConstants.HTTP_CLI
      ENT_INCOMING_CONTEXT, true);

6  String contentType = httpAttribs.getContentType();
```

The code in Example 215 does the following:

1. Imports the package containing the HTTP client context type.
2. Makes an invocation on the proxy.

3. Gets the client's context registry.
4. Gets the Artix context from the context registry.
5. Gets the client's incoming HTTP context from the reply context container.
6. Gets the value of the `ContextType` property.

Incoming client attributes

Table 17 shows the attributes that are valid in the incoming HTTP client context.

Table 17: *Incoming HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
`Content-Encoding`	`String getContentEncoding()`	Specifies the type of special encoding, if any, the server used to package the response.
`Content-Language`	`String getContentLanguage()`	Specifies the language the server used in writing the response. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, `en-US`.
`Content-Location`	`String getContentLocation()`	Specifies the URL where the resource being sent in a response is located.
`Content-Type`	`String getContentType()`	Specifies the MIME type of the data in the response.
`ETag`	`String getETag()`	Specifies the entity tag in the response header.
`HTTPReply`	`String getHTTPReply()`	Specifies the type of reply being sent back by the server. For example, if a request is fulfilled a server will reply with `OK`.
`HTTPReplyCode`	`Integer getHTTPReplyCode()`	Specifies an integer code associated with the server's reply. For example, `200` means `OK` and `404` means `Not Found`.

Table 17: *Incoming HTTP Client Attributes*

HTTP Attribute	Artix APIs	Description
`Last-Modified`	`String getLastModified()`	Specifies the date and time at which the server believes a resource was last modified.
`Proxy-Authenticate`	`String getProxyAuthenticate()`	Specifies a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.
`RedirectURL`	`String getRedirectURL()`	Specifies the URL to which client requests should be redirected. This is issued by a server when it is not appropriate for the request.
`ServerType`	`String getServerType()`	Specifies the type of server responded to the client. Values take the form *`program-name/version`*.
`WWW-Authenticate`	`String getWWWAuthentication()`	Specifies at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

Server-side Configuration

Overview

HTTP servers have access to both the values being passed in the HTTP header of the outgoing response and the values received in the HTTP header of the request. The information for each header is stored in a separate context.

Outgoing header

On the server-side, the outgoing context, `HTTP_SERVER_OUTGOING_CONTEXT`, is available in the server's reply context container. Any changes made to values in the outgoing context are placed in the reply's HTTP header and propagated to the client. For example, if you want to inform the client that it needs to redirect it's request to a different server, you could set the `RedirectURL` attribute in the server's outgoing context to the URL of an appropriate server. Example 216 shows the code for setting the `RedirectURL` attribute for a server.

Example 216: *Setting a Server's RedirectURL Attribute*

```
1  import com.iona.schemas.transports.http.configuration.context.*;
   import com.iona.jbus.ContextConstants;

   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  ClientType httpAtribs =
      (ClientType)context.getReplyContext(ContextConstants.HTTP_SER
      VER_OUTGOING_CONTEXT, true);

5  httpAtribs.setRedirectURL("http:\\www.notme.org\askthisguy");
```

The code in Example 216 does the following:

1. Imports the package containing the HTTP server context type.
2. Gets the server's context registry.
3. Gets the Artix context from the context registry.
4. Gets the server's outgoing HTTP context from the reply context container.

5. Sets the value of the `RedirectURL` property to the URL of the server who can satisfy the request.

Outgoing server attributes

Table 18 shows the attributes that are valid in the outgoing HTTP server context.

Table 18: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Cache-Control`	`String getCacheControl()` `void setCacheControl(String val)`	Specifies directives to caches along the request/response path. Valid values are: **`no-cache`**: caches must revalidate responses with the server. If response header fields are given, the restriction applies only to those header fields. **`public`**: any cache can store the response. **`private`**: public caches cannot store the response. If response header fields are given, the restriction applies only to those header fields. **`no-store`**: caches must not store any part of the response or the request. **`no-transform`**: caches must not modify the media type or the content location of a response.

Table 18: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
		must-revalidate: caches must revalidate responses that have expired with the server before the response can be used. **proxy-revalidate**: means the same as must-revalidate, but it can only be enforced on shared caches. You must set the public directive when using this directive. **max-age**: the max age, in seconds, of an acceptible response. **s-maxage**: means the same as max-age, but it can only be enforced on shared caches. When set it overides the value of max-age. You must use the proxy-revalidate directive when using this directive.
Content-Encoding	String getContentEncoding() void setContextEncoding(String val)	Specifies the type of special encoding, if any, the server uses to package a response.
Content-Language	String getContentLanguage() void setContentLanguage(String val)	Specifies the language used to write a response. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, en-US.
Content-Location	String getContentLocation() void setContentLocation(String val)	Specifies the URL where the resource being sent in a response is located.
Content-Type	String getContentType() void setContentType(String val)	Specifies the MIME type of the data in the response.

Table 18: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`ETag`	`String getETag()` `void setETag(String val)`	Specifies the entity tag in the response header.
`Expires`	`String getExpires()` `void setExpires(String val)`	Specifies the date after which the response is considered stale.
`HonorKeepAlive`	`Boolean isHonorKeepAlive()` `void setHonorKeepAlive(Boolean val)`	Specifies if the server is going to honor a client's keep-alive request.
`HTTPReply`	`String getHTTPReply()` `void setHTTPReply(String val)`	Specifies the type of response the server is issuing. For example, if the request is fulfilled the server will reply with `OK`.
`HTTPReplyCode`	`Integer getHTTPReplyCode()` `void setHTTPReplyCode(Integer val)`	Specifies an integer code associated with the response. For example, `200` means `OK` and `404` means `Not Found`.
`Last-Modified`	`String getLastModified()` `void setLastModified(String val)`	Specifies the date and time at which the server believes a resource was last modified.
`Pragma`	`String getPragma()` `void setPragma(String val)`	Specifies implementation-specific directives that might apply to any recipient along the request/response chain.

`Proxy-Authorization`	`String getProxyAuthroization()`
	`void setProxyAuthentication(`
	`String val)`
		Specifies the credentials used to perform validation at a proxy server along the request/response chain. If the proxy uses username/password validation, this value is not used.

`ProxyAuthorizationType`	`String getProxyAuthorizationType()`
	`void setProxyAuthorizationType(`
	`String val)`
		Specifies the type of authentication used by proxy servers along the request/response chain.

Table 18: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`ProxyPassword`	`String getProxyPassword()` `void setProxyPassword(String val)`	Specifies the password used by proxy servers for authentication if username/password authentication is in use.
`ProxyServer`	`String getProxyServer()` `void setProxyServer(String val)`	Specifies the URL of the proxy server, if one exists, along the request/response chain. **Note:** Artix does not support the existence of more than one proxy server along the request/response chain.
`ProxyUserName`	`String getProxyUsername()` `void setProxyUserName(String val)`	Specifies the username used by proxy servers for authentication if username/password authentication is in use.
`Recieve-Timeout`	`Integer getRecieveTimeout()` `void setRecieveTimeout(Integer val)`	Specifies the number of milliseconds the server will wait to receive a request before timing out. The default is `3000`.
`RedirectURL`	`String getRedirectURL()` `void setRedirectURL(String val)`	Specifies the URL to which the request should be redirected.
`Send-Timeout`	`Integer getSendTimeout()` `void setSendTimeout(Integer val)`	Specifies the number of milliseconds the server will continue trying to send a response before timing out. The default is `3000`.
`ServerCertificate`	`String getServerCertificate()` `void setServerCertificate(String val)`	Specifies the full path to the X509 certificate issued by the certificate authority for the server.
`ServerCertificateChain`	`String getServerCertificateChain()` `void setServerCertificateChain(String val)`	Specifies the full path to the file containing all of the certificates in the chain.

Table 18: *Outgoing HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Server Type`	`String getServerType()` `void setServerType(String val)`	Specifies the type of server responded to the client. Values take the form *`program-name/version`*.
`ServerPrivateKey`	`String getServerPrivateKey()` `void setServerPrivateKey(String val)`	Specifies the full path to the PKCS12-encoded private key that corresponds to the X509 certificate specified by `ServerCertificate`.

`ServerPrivateKeyPassword`	`String getServerPrivateKeyPassword()`
	`void getServerPrivateKeyPassword(`
	`String val)`
		Specifies the password used to decrypt the PKCS12-encoded private key.

`Trusted Root Certificate`	`String getTrustedRootCertificates()`
	`void setTrustedRootCertificates(`
	`String val)`
		Specifies the full path to the PKCS12-encoded X509 certificate for the certificate authority.

`UseSecureSockets`	`Boolean isUseSecureSockets()` `void setUseSecureSockets(Boolean val)`	Specifies the server wants to use a secure connection. Secure HTTP connections are also referred to as HTTPS. **Note:** If the contract specifies HTTPS, this value is always `true`.
`WWW-Authenticate`	`String getWWWAuthentication()` `void setWWWAunthentication(String val)`	Specifies at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

Incoming header

The server's incoming context, `HTTP_SERVER_INCOMING_CONTEXT`, is available in the server's request context container after a request from client has been received by the transport layer. The values stored in this context are for

informational purposes only. For example, if you need to check the MIME type of the data the client can accept in the response, you would read it from the server's incoming context as shown in Example 217.

Example 217: *Reading the Accept Attribute in an HTTP Server*

```
1  import com.iona.schemas.transports.http.configuration.context.*;
   import com.iona.jbus.ContextConstants;

   ...
2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  ClientType httpAtribs =
      (ClientType)context.getRequestContext(ContextConstants.HTTP_S
      ERVER_INCOMING_CONTEXT, true);

5  String contentType = httpAttribs.getAccept();
```

The code in Example 217 does the following:

1. Imports the package containing the HTTP server context type.
2. Gets the server's context registry.
3. Gets the Artix context from the context registry.
4. Gets the server's incoming HTTP context from the reply context container.
5. Gets the value of the `Accept` attribute.

Incoming server attributes

Table 17 shows the attributes that are valid in the incoming HTTP server context.

Table 19: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Accept`	`String getAccept()`	Specifies the MIME types the client can handle in a response.

Table 19: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Accept-Encoding`	`String getAcceptEncoding()`	Specifies the types of content encoding the client can handle in a response. This property typically refers to compression mechanisms.
`Accept-Language`	`String getAcceptLanguage()`	Specifies the language preferred by the client. Valid language tags combine an ISO language code and an ISO country code separated by a hyphen. For example, `en-US`.
`Authorization`	`String getAuthorization()`	Specifies the credentials that will be used by the server to authorize requests from the client.
`AuthorizationType`	`String getAuthorizationType()`	Specifies the name of the authentication scheme in use.
`AutoRedirect`	`Boolean isAutoRedirect()`	Specifies whether the server should automatically redirect the request.
`BrowserType`	`String getBrowserType()`	Specifies information about the browser from which the request originates. This property is also know as the user-agent.
`Certificate Issuer`	`String getCertificateIssuer()`	Specifies the value stored in the `Issuer` field of the client's X509 certificate.
`Certificate Key Size`	`Integer getCertificateKeySize()`	Specifies the size, in bytes, of the public key included in the client's x509 certificate.
`Certificate Valid Not After`	`String getCertificateNotAfter()`	Specifies the date and time after which the client's X509 certificate is invalid.

Table 19: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Certificate Valid Not Before`	`String getCertificateNotBefore()`	Specifies the date and time before which the client's X509 certificate is invalid.
`Certificate Subject`	`String getCertificateSubject()`	Specifies the value of the `Subject` field in the client's X509 certificate.
`Connection`	`String getConnection()`	Specifies whether a connection is to be kept open after each request/response transaction.
`Cookie`	`String getCookie()`	Specifies a static cookie that is sent along with a request. **Note:** According to the HTTP 1.1 specification, HTTP cookies must contain US-ASCII characters.
`Host`	`String getHost()`	Specifies the Internet host and port number of the resource being requested.
`HTTPVersion`	`String getHTTPVersion()`	Specifies the version of the HTTP transport in use. Currently, this is always set to 1.1.
`If-Modified-Since`	`String getIfModifiedSince()`	If the requested resource has not been modified since the time specified, the server should issue a 304 (not modified) response without any message body.
`Method`	`String getMethod()`	Specifies the value of the `METHOD` token sent in the request. Valid values and their meanings are given in the HTTP 1.1 specification.
`Passwrod`	`String getPassword()`	Specifies the password the client wishes to use for authentication.

Table 19: *Incoming HTTP Server Attributes*

HTTP Attribute	Artix APIs	Description
`Proxy-Authenticate`	`String getProxyAuthenticate()`	Specifies a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.
`Referer`	`String getReferer()`	Specifies the entity that referred the client.
`URL`	`String getURL()`	Specifies the value of the Request-URI sent in the request. The valid values for this property are described in the HTTP 1.1 specification.
`Username`	`String getUserName()`	Specifies the username the client wishes to use for authentication.

Setting the Server's Endpoint URL

Overview

Because the server's endpoint URL must be known before the transport layer is initialized by the bus, you must use the specialized configuration context to set it. For more information on using the configuration context see "Using the Configuration Context" on page 312.

Getting the property

To access the HTTP endpoint URL property for an HTTP server, you use the `ContextConstants` member `HTTP_SERVER_OUTGOING_CONTEXTS`. You are returned a `ServerType` object that has two relevant methods:

- `setURL()` sets a `String` representing the URL of the server.
- `getURL()` returns a `String` representing the URL of the server.

Example

Example 218 shows how to set the HTTP Endpoint URL programatically.

Example 218: *Setting the HTTP Endpoint URL*

```
1 ContextRegistry registry = bus.getContextRegistry();

2 QName name = new QName("http://www.iona.com/config_context",
                         "SOAPService");

3 ContextContainer contain = registry.getConfigurationContext(
                                                          name,
                                                    "SoapPort",
                                                         true);

4 ServerType httpConf = (ServerType)container.getContext(
                    ContextConstants.HTTP_SERVER_OUTGOING_CONTEXTS,
                      true);

5 httpConf.setURL("http://localhost:63278/config_context_test");

  ...
6 bus.registerServant(servant, qname, portName);
```

The code in Example 218 does the following:

1. Get the context registry.
2. Create the service's `QName`.
3. Get the configuration context container.
4. Get the server's outgoing HTTP context.
5. Set the endpoint URL property.
6. Register the servant.

Setting CORBA Attributes

Overview

The CORBA transport does not support programmatic configuration. It also does not provide access to any of the settings that are used to establish the connection. Artix does, however, provide access to the CORBA principle by way of the context mechanism. The CORBA principle is manipulated as a `String` by the Java contexts.

Retrieving the CORBA principle

Generally, you would only be inspecting the CORBA principle of an incoming message. This means that in an Artix server, you would get the CORBA context from the Artix request context container. In an Artix client, you would get the CORBA context from the Artix reply context container.

Example 219 shows the code for getting the CORBA principle in a server.

Example 219: *Getting the CORBA Principle from a Client's Request*

```
1  import com.iona.schemas.bindings.corba.contexts.*;
   import com.iona.jbus.ContextConstants;

   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  CORBAAttributesType CORBAAtribs =
      (CORBAAttributesType)context.getRequestContext(ContextConstan
      ts.CORBA_CONTEXT_ATTRIBUTES, true);

5  String CORBAPrinciple = CORBAAtribs.getPrinciple();
```

The code in Example 219 does the following:

1. Imports the package containing the CORBA context type.
2. Gets the server's context registry.
3. Gets the Artix context from the context registry.
4. Gets the server's CORBA context from the request context container.
5. Gets the principle.

Setting the CORBA principle

The CORBA principle is typically used for interoperability with older CORBA servers to set security information. In most cases, you would set the CORBA principle in a client's request message using the client's request context. You can also set the CORBA principle in a server's reply message using the server's reply context.

Example 220 shows the code for setting the CORBA principle for a client request.

Example 220: *Setting the CORBA Principle for a Client's Request*

```
1  import com.iona.schemas.bindings.corba.contexts.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  CORBAAttributesType CORBAAtribs =
      (CORBAAttributesType)context.getRequestContext(ContextConstan
      ts.CORBA_CONTEXT_ATTRIBUTES, true);

5  String username = new String("Fred");
6  CORBAAtribs.setPrinciple(username);

7  // Make invocation on proxy
```

The code in Example 219 does the following:

1. Imports the package containing the CORBA context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the CORBA context from the request context container.
5. Creates a new `String` to hold the value to set into the CORBA principle.
6. Sets the principle.
7. Make the invocation on the proxy.

Setting WebSphere MQ Attributes

Overview

When working with WebSphere MQ, your applications can access information about the WebSphere MQ connection that is in use and information contained in the WebSphere MQ message descriptor. The MQ connection attributes context contains information about the queues and queue managers that your application uses for send and receiving messages. On the client-side, you can set this information on a per-invocation basis. The MQ message attributes context allows you to inspect and set a number of the properties stored in the WebSphere MQ message descriptor.

In this section

This section discusses the following topics:

Working with Connection Attributes

Overview

The WebSphere MQ transport provides information about the queues to which your application send and receives messages. This information is stored in the MQ connection attributes context and is accessed using `ContextConstants.MQ_CONNECTION_ATTRIBUTES`. The data is returned in an `MQConnetionAttributesContextType` object. Table 20 describes the attributes stored in the MQ connection attributes context.

Table 20: *MQ Connection Attributes Context Properties*

Attribute	Artix APIs	Description
AliasQueueName	`String getAliasQueueName()` `void setAliasQueueName(String val)`	Specifies the remote queue to which a server will put replies if its queue manager is not on the same host as the client's local queue manager.
ConnectionName	`String getConnectionName()` `void setConnecitonName(String val)`	Specifies the name of the connection by which the adapter connects to the queue.
ModelQueueName	`String getModelQueueName()` `void setModelQueueName(String val)`	Specifies the name of the queue to be used as a model for creating dynamic queues.
QueueManager	`String getQueueManager()` `void setQueueManager(String val)`	Specifies the name of the queue manager.
QueueName	`String getQueueName()` `void setQueueName(String val)`	Specifies the name of the message queue.
ReplyQueueManager	`String getReplyQueueManager()` `void setReplyQueueManager(String val)`	Specifies the name of the reply queue manager. This setting is ignored by WebSphere MQ servers when the client specifies the `ReplyToQMgr` in the request message's message descriptor.

Table 20: *MQ Connection Attributes Context Properties*

Attribute	Artix APIs	Description
ReplyQueueName	`String getReplyQueueName()` `void setReplyQueueName(String val)`	Specifies the name of the queue where response messages are received. This setting is ignored by WebSphere MQ servers when the client specifies the `ReplyToQ` in the request message's message descriptor.
Transactional	`TransactionType getTransactional()` `void setTransactional(TransactionType val)`	Specifies how messages participate in transactions and what role WebSphere MQ plays in the transactions. For information on setting Transactional see "Setting the Transactional attribute" on page 341.

On the client-side you can control the connection to which requests are direct by setting the MQ connection attributes in the client's request context before each invocation. The connection attributes are returned to the defaults specified in the client's contract after each invocation.

Example

Example 221 shows code for specifying the queue and queue manager to use when making a request.

Example 221:*Setting the Client's QueueManager and QueueName*

```
1  import com.iona.schemas.transports.mq.context.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  MQConnectionAttributesType connect =
      (MQConnectionAttributesType)context.getRequestContext(Context
      Constants.MQ_CONNECTION_ATTRIBUTES, true);
```

Example 221: *Setting the Client's QueueManager and QueueName*

```
5  connect.setQueueManager("Bloggy");
6  connect.setQueueName("TalkBack");

7  // Make invocation on proxy
```

The code in Example 221 does the following:

1. Imports the package containing the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the MQ connection attributes context from the request context container.
5. Sets the queue manager attribute.
6. Sets the queue name attribute.
7. Makes the invocation on the proxy.

On the server-side you cannot change any of the connection attributes programmatically.

Setting the Transactional attribute

The transactional attribute is set using a `com.iona.schemas.transports.mq.context.TransactionType` object. `TransactionType` is a WSDL enumeration whose values are described in Table 21.

Table 21: *Transactional Values*

Value	Artix API for Setting	Description
`none`	`setTransactional(TransactionType.fromString("none"))`	The messages are not part of a transaction. No rollback actions will be taken if errors occur.
`internal`	`setTransactional(TransactionType.fromString("internal"))`	The messages are part of a transaction with WebSphere MQ serving as the transaction manager.

Table 21: *Transactional Values*

Value	Artix API for Setting	Description
xa	`setTransactional(TransactionType.fromString("xa"))`	The messages are part of a transaction with WebSphere MQ serving as the resource manager.

Example 222 shows code for setting a client's connection to use XA style transactionality for a request.

Example 222: *Setting the Client's Transactionality Attribute*

```
1  import com.iona.schemas.transports.mq.context.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  MQConnectionAttributesType connect =
      (MQConnectionAttributesType)context.getRequestContext(Context
      Constants.MQ_CONNECTION_ATTRIBUTES, true);

5  connect.setTransactional(TransactionType.fromString("xa"));

6  // Make invocation on proxy
```

The code in Example 221 does the following:

1. Imports the package containing the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the MQ connection attributes context from the request context container.
5. Sets the transactional attribute.
6. Makes the invocation on the proxy.

For more information about working with Artix enumerated types, see "Using Enumerations" on page 71.

Working with MQ Message Descriptor Attributes

Overview

The Artix WebSphere MQ transport breaks its support for MQ message descriptor attributes across two contexts. One context, accessed using `ContextConstants.MQ_INCOMING_MESSAGE_ATTRIBUTES`, contains the MQ message descriptor attributes for the last message received by the application. For a client, this means that it contains the attributes for the last response received from the server and the context is accessed through the client's reply context container. For a server, this means that the incoming message attributes context contains the descriptor attributes for the request being processed and it is accessed through the server's request context container. The incoming message properties can be read at any point in the processing of the message once the transport layer has passed it to the messaging chain.

The second context, accessed using `ContextConstants.MQ_OUTGOING_MESSAGE_ATTRIBUTES`, allows you to set the values of the attributes in the MQ message descriptor for the next message being sent across the wire. For clients, this means that it affects the values of the next request being made and the context is accessed through the client's request context. For server's, this means that the outgoing message attributes context affects the values of the current response's MQ message descriptor and it is accessed through the server's reply context container. You can set the values of the outgoing message attributes at any point in an application's message chain before it the message is handed off to the transport layer.

Both the incoming message attributes context and the outgoing message attributes context are returned using as an `com.iona.schemas.transports.mq.context.MQMessageAttributesType` object. Table 22 describes the attributes stored in the MQ message attributes context.

Table 22: *MQ Message Attributes Context Properties*

Attribute	Artix APIs	Description
AccountingToken	`String getAccountingToken()` `void setAccountingToken(String val)`	Specifies the value for the MQ message decscriptor's `AccountingToken` field.

Table 22: *MQ Message Attributes Context Properties*

Attribute	Artix APIs	Description
ApplicationData	`String getApplicationData()` `void setApplicationData(String val)`	Specifies any application-specific information that needs to be set in the message descriptor.
ApplicationIdData	`String getApplicationIdData()` `void setApplicationIdData(String val)`	Specifies the value of the MQ message descriptor's `ApplIdentityData` field. It is only valid for MQ clients.
ApplicationOriginData	`String getApplicationOriginData()` `void setApplicationOriginData(String val)`	Specifies the value of the MQ message descriptor's `ApplOriginData` field.
BackoutCount	`Integer getBackoutCount()`	Specifies the number of times the message has been previously returned by the `MQGET` call as part of a unit of work, and subsequently backed out.
Convert	`Boolean isConvert()` `void setConvert(Boolean val)`	Specifies if the messages in the queue needs to be converted to the system's native encoding.
CorrelationId	`byte[] getCorrelationId()` `void setCorrelationId(byte[] val)`	Specifies the value for the MQ message descriptor's `CorrelId` field.

CorrelationStyle	`CorrelationStyleType getCorrelationStyle()`
	`void`
	`  setCorrelationStyle(CorrelationStyleType`
	`  val)`
		Specifies how WebSphere MQ matches both the message identifier and the correlation identifier to select a particular message to be retrieved from the queue. For information on how to set CorrelationStyle, see "Setting the CorrelationStyle attribute" on page 346.

Table 22: *MQ Message Attributes Context Properties*

Attribute	Artix APIs	Description
Delivery	`DeliveryType getDelivery()` `void setDelivery(DeliveryType val)`	Specifies the value of the MQ message descriptor's `Persistence` field. For information on setting Delivery, see "Setting the Delivery attribute" on page 347.
Format	`FormatType getFormat()` `void setFormat(FormatType val)`	Specifies the value of the MQ message descriptor's `Format` field. For information on setting Format, see "Setting the Format attribute" on page 348.
MessageId	`byte[] getMessageId()` `void setMessageId(byte[] val)`	Specifies the value for the MQ message descriptor's `MsgId` field.
ReportOption	`ReportOptionType getReportOption()` `void setReportOption(ReportOptionType val)`	Specifies the value of the MQ message descriptor's `Report` field. For information on setting ReportOption, see "Setting the ReportOption attribute" on page 350.
UserIdentifier	`String getUserIdentifier()` `void setUserIdentifier(String val)`	Specifies the value for the MQ message descriptor's `UserIdentifier` field.

Setting the CorrelationStyle attribute

The CorrelationStyle attribute is set using a `com.iona.schemas.transports.mq.context.CorrealatoinStyleType` object. `CorrelationStyleType` is a WSDL enumeration whose values are described in Table 23.

Table 23: *CorrelationStyle Values*

Value	Artix API for Setting	Description
`messageId`	`setCorrelationStyle( CorrelationStyleType.fromString("messageId") )`	Use the message ID as the value for the message's `CorrelId`.
`correlationId`	`setCorrelationStyle( CorrelationStyleType.fromString("correlationId") )`	Use the message's CorrelationId as the value for the message's `CorrelId`.
`messageId copy`	`setCorrelationStyle( CorrelationStyleType.fromString("messageId_copy") )`	Use the message ID as the value for the message's `MsgId`.

Example 223 shows code for setting a request message descriptor's CorrelationStyle message Id.

Example 223: *Setting the Client's CorrelationStyle Attribute*

```
1  import com.iona.schemas.transports.mq.context.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  MQMessageAttributesType desc =
      (MQMessageAttributesType)context.getRequestContext(ContextCon
      stants.MQ_OUTGOING_MESSAGE_ATTRIBUTES, true);

5  connect.setCorrelationStyle(
     CorrelationStyleType.fromString("messageId")
     );

6  // Make invocation on proxy
```

The code in Example 223 does the following:

1. Imports the package containing the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the MQ connection attributes context from the request context container.
5. Sets the correlation style attribute.
6. Makes the invocation on the proxy.

For more information about working with Artix enumerated types, see "Using Enumerations" on page 71.

Setting the Delivery attribute

The Delivery attribute is set using a `com.iona.schemas.transports.mq.context.DeliveryType` object. `DeliveryType` is a WSDL enumeration whose values are described in Table 24.

Table 24: *Delivery Values*

Value	Artix API for Setting	Description
`persistent`	`setDelivery(DeliveryType.fromString("persistent"))`	Sets the `Persistence` field to `MQPER_PERSISTENT`.

`not persistent`	`setDelivery(`
	`  DelvieryType.fromString("not_persistent")`
	`  )`
		Sets the `Persistence` field to `MQPER_NOT_PERSISTENT`.

Example 224 shows code for setting a request message descriptor's `Persistence` field to `MQPER_PERSISTENT`.

Example 224: *Setting the Client's Delivery Attribute*

```
1  import com.iona.schemas.transports.mq.context.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
```

Example 224: *Setting the Client's Delivery Attribute*

```
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  MQMessageAttributesType desc =
      (MQMessageAttributesType)context.getRequestContext(ContextCon
      stants.MQ_OUTGOING_MESSAGE_ATTRIBUTES, true);

5  connect.setDelivery(DeliveryType.fromString("persistent"));

6  // Make invocation on proxy
```

The code in Example 224 does the following:

1. Imports the package containing the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the MQ connection attributes context from the request context container.
5. Sets the delivery attribute.
6. Makes the invocation on the proxy.

For more information about working with Artix enumerated types, see "Using Enumerations" on page 71.

Setting the Format attribute

The Format attribute is set using a `com.iona.schemas.transports.mq.context.FormatType` object. `FormatType` is a WSDL enumeration whose values are described in Table 25.

Table 25: *Format Values*

Value	Artix API for Setting	Description
none	`setFormat(FormatType.fromString("none"))`	Sets the `Format` field to `MQFMT_NONE`.
string	`setFormat(FormatType.fromString("string"))`	Sets the `Format` field to `MQFMT_STRING`.

Table 25: *Format Values*

Value	Artix API for Setting	Description
unicode	setFormat(FormatType.fromString("unicode"))	Sets the Format field to MQFMT_STRING.
event	setFormat(FormatType.fromString("event"))	Sets the Format field to MQFMT_EVENT.
programmable command	setFormat(FormatType.fromString("programmable_command"))	Sets the Format field to MQFMT_PCF.

Example 225 shows code for setting a request message descriptor's Format field to MQPER_STRING.

Example 225: *Setting the Client's Format Attribute*

```
1  import com.iona.schemas.transports.mq.context.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  MQMessageAttributesType desc =
      (MQMessageAttributesType)context.getRequestContext(ContextCon
      stants.MQ_OUTGOING_MESSAGE_ATTRIBUTES, true);

5  connect.setFormat(FormatType.fromString("string"));

6  // Make invocation on proxy
```

The code in Example 225 does the following:

1. Imports the package containing the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the MQ connection attributes context from the request context container.
5. Sets the format attribute.

6. Makes the invocation on the proxy.

For more information about working with Artix enumerated types, see "Using Enumerations" on page 71.

Setting the ReportOption attribute

The ReportOption attribute is set using a `com.iona.schemas.transports.mq.context.ReportOptionType` object. `ReportOptionType` is a WSDL enumeration whose values are described in Table 26.

Table 26: *ReportOption Values*

Value	Artix API for Setting	Description
`coa`	`setReportOption(ReportOption.fromString("coa"))`	Set the message descriptor's `Report` field to `MQRO_COA`.
`cod`	`setReportOption(ReportOption.fromString("cod"))`	Set the message descriptor's `Report` field to `MQRO_COD`.

`exception`	`setReportOption(`
	`  ReportOption.fromString("exception")`
	`  )`
		Set the message descriptor's `Report` field to `MQRO_EXCEPTION`.

`expiration`	`setReportOption(`
	`  ReportOption.fromString("expiration")`
	`  )`
		Set the message descriptor's `Report` field to `MQRO_EXPIRATION`.

`discard`	`setReportOption(ReportOption.fromString("discard")` `  )`	Set the message descriptor's `Report` field to `MQRO_DISCARD_MSG`.

Example 226 shows code for setting a request message descriptor's `Report` field to `MQRO_DISCARD_MSG`.

Example 226: *Setting the Client's ReportOption Attribute*

```
1  import com.iona.schemas.transports.mq.context.*;
   import com.iona.jbus.ContextConstants;
   ...

2  ContextRegistry contReg = bus.getContextRegistry();
```

Example 226: *Setting the Client's ReportOption Attribute*

```
3  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

4  MQMessageAttributesType desc =
      (MQMessageAttributesType)context.getRequestContext(ContextCon
      stants.MQ_OUTGOING_MESSAGE_ATTRIBUTES, true);

5  connect.setReportOption(ReportOptionType.fromString("discard"));

6  // Make invocation on proxy
```

The code in Example 226 does the following:

1. Imports the package containing the MQ connection attributes context type.
2. Gets the client's context registry.
3. Gets the Artix context from the context registry.
4. Gets the MQ connection attributes context from the request context container.
5. Sets the report option attribute.
6. Makes the invocation on the proxy.

For more information about working with Artix enumerated types, see "Using Enumerations" on page 71.

Setting JMS Attributes

Overview

When using JMS, Artix splits the JMS transport information into three contexts:

- one for JMS clients.
- one for JMS servers.
- one to register JMS enabled Artix applications with a secure JMS broker.

The JMS server context and the JMS client context provide access to the JMS message header attributes. It includes information about message expiration, message persistence, message correlation, and when the message was created. In addition, the JMS header contexts enable you to set optional properties into the JMS header for use with message selectors.

Both the JMS server context and the JMS client context provide access to specific properties that alter the behavior of the transport. For instance the JMS client context allows you to specify a timeout value for messages.

In this section

This section discusses the following topics:

Using JMS Message Headers and Properties

Overview

A JMS message is composed of three sections:

- a JMS header containing a number of standard properties effecting ho a message is handled.
- a group of name/value properties that specify optional information about the message.
- the message body.

Using the context mechanism, Artix allows you to inspect all members of the JMS header. It also allows you to set the values for members that are not set by the JMS broker. In addition, the context mechanism provides you with a way to set properties into the properties group of the JMS message.

Standard JMS attributes available from the context

Table 27 shows the JMS header attributes available for both the JMS client context and the JMS server context. Not all of the JMS header attributes are settable. For those that are settable, both the getter and the setter methods are shown.

Table 27: *JMS Header Attributes*

JMS Header Attribute	Artix API	Description
JMSCorrelationID	`String getJMSCorrelationID()`	Specifies the message's correlation ID.
JMSDeliveryMode	`Integer getJMSDeliveryMode()` `void setJMSDeliveryMode(Integer val)`	Specifies if the message is persistent or non-persistent. Valid values are `PERSITENT` and `NON_PERSISTENT`. The default is `PERSISTENT`.
JMSExpiration	`Long getJMSExpiration()`	Specifies the time at which the message expires. An expiration of `0` means that the message never expires.
JMSMessageID	`String getJMSMessageID()`	Specifies the unique ID assigned to the message by the JMS broker.

Table 27: *JMS Header Attributes*

JMS Header Attribute	Artix API	Description
JMSPriority	`Integer getJMSPriority()` `void setJMSPriority(Integer val)`	Specifies the relative priority of the message. Valid values are `0-9`. `0` is the lowest priority. The default priority is `4`.
Optional Properties	`JMSPropertyType[] getProperty()` `void setProperty(JMSPropertyType[] val)`	Specifies any number of user-defined properties that are used in conjunction with JMS message selectors.
JMSRedelivered	`Boolean isJMSRedelivered()`	Specifies if the JMS broker believes that this message has already been delivered, but not acknowledged.
JMSTimestamp	`Long getJMSTimeStamp()`	Specifies the time at which the message was handed off to the JMS broker.
JMSType	`String getJMSType()`	Specifies the type of the message. Some JMS implementations use this field to specify templates for messages.
Time To Live	`Long getTimeToLive()` `void setTimeToLive(Long val)`	Specifies the number of milliseconds the message will remain active in the JMS destination to which it is delivered. The default value is unlimited.

Creating optional JMS header properties

A part of the JMS header is set aside for optional properties. These properties include a few standard properties that are prefixed with JMSX. JMS vendors also use the properties section of the JMS message to specify vendor-specific information. The properties section can also be used as a place to store user-defined properties that can be used for message selection among other things.

The JMS properties are stored in the JMS header as name value pairs. In Artix JMS properties are created in `com.iona.schemas.transports.jms.context.JMSProperyType` objects. `JMSProperty` objects have two members and getter and setter methods for each member. The `name` member specifies the name by which the property will be referred. It can be any string value. The `value` member stores the data of the property and can also be any string value.

Properties are set into the JMS header using the outbound JMS context's `setProperty()` method. `setProperty()` takes an array of properties, so you can create as many user-defined properties as you wish.

Example 227 shows how to create a set of user-defined properties and set them on a client request's JMS message.

Example 227: *Creating User-Defined Properties and Setting Them into a JMS Header*

```
1  import com.iona.schemas.transports.jms.context.*;
   import com.iona.jbus.ContextConstants;

2  JMSPropertyType[] props = new JMSPropertyType[2];

3  props[0] = new JMSPropertType();
   props[0].setName("Username");
   props[0].setValue("Flint");

4  props[1] = new JMSPropertType();
   props[1].setName("Password");
   props[1].setValue("Moore");

5  ContextRegistry contReg = bus.getContextRegistry();
6  IonaMessageContext context =
    (IonaMessageContext)contReg.getCurrent();

7  JMSClientHeadersType header =
      (JMSClientHeadersType)context.getRequestContext(ContextConsta
      nts.JMS_CLIENT_CONTEXT, true);

8  header.setProperty(props);

9  // Make invocation on proxy
```

The code in Example 227 does the following:

1. Imports the package containing the JMS context types.
2. Creates an array of two `JMSPropertyType` objects to hold the user-defined properties.
3. Sets the name/value pair for the first property.
4. Sets the name/value pair for the second property.
5. Gets the client's context registry.
6. Gets the Artix context from the context registry.
7. Gets the JMS context from the request context container.
8. Sets the user-defined properties into the JMS context.
9. Makes an invocation on the proxy.

Using Client-side JMS Attributes

Overview

When working with JMS clients you get the JMS header information using the JMS client context which is accessed using the `JMS_CLIENT_CONTEXT` tag. The JMS client context information is returned as a `JMSClientHeadersType` object. The JMS client context has all of the standard JMS header attributes plus an additional `TimeOut` attribute.

Timeout

The `Timeout` attribute specifies the value passed into the JMS message consumer's `recieve()` method. The time-out value is specified as a `Long` and determines how long, in milliseconds, the message consumer will wait for a message to arrive before timing out. Example 228 shows the methods for accessing the `TimeOut` value on a `JMSClientHeadersType` object.

Example 228: *Methods for Accessing the TimeOut Value*

```
Long getTimeOut();
void setTimeOut(Long timeout);
```

Setting the client attributes

Most of the attributes in the JMS header are populated by the JMS broker and are provided simply for informational purposes. However, when making requests you can add any number of user-defined properties to the header as shown in "Creating optional JMS header properties" on page 354. In addition, you can set the message's `JMDDeliveryMode`, the message's `JMSPriority`, the message's time to live, and the time-out interval used to wait for a response. To set these properties, you use the JMS client context from the client's request context container at any point along the messaging chain before the message is handed off to the transport layer. The settable attributes are valid for one request and are reset once the request is sent to the JMS broker.

To set the user settable JMS client attributes do the following:

1. Get the application's message context.
2. Get the JMS client context from the request context container.
3. Set the desired property values on the JMS client context.

Example 229 shows the code for setting the JMS client attributes for a request.

Example 229:*Setting a Request's JMS Header Attributes*

```
import com.iona.schemas.transports.jms.context.*;
import com.iona.jbus.ContextConstants;

1  IONAMessageContext cont = (IONAMessageContext)
     DispatchLocals.getCurrentMessageContext();

2  JMSClientHeadersType header = (JMSClientHeadersType)
     cont.getRequstContext(ContextConstants.JMS_CLIENT_CONTEXT,
                           true);

3  header.setJMSDeliveryMode("NON_PERSISTENT");
   header.setJMSPriority(new Integer(7));
   header.setTimeToLive(new Long(120000));
   header.setTimeOut(new Long(3000));

   // Make invocation on proxy
```

Inspecting the client attributes

To inspect the JMS header values of a response message, you get the JMS client context from the client's reply context container. The values in the context are valid for the last response received from the server. They are available once the transport layer passes the message up the messaging chain.

Example 230 shows code for checking the `JMSCorrelationID` of a response.

Example 230:*Checking a Responses JMSCorrelationID*

```
import com.iona.schemas.transports.jms.context.*;
import com.iona.jbus.ContextConstants;

// Make invocation on proxy

IONAMessageContext cont = (IONAMessageContext)
  DispatchLocals.getCurrentMessageContext();

JMSClientHeadersType header = (JMSClientHeadersType)
  cont.getReplyContext(ContextConstants.JMS_CLIENT_CONTEXT,
                       true);

String corrID = header.getJMSCorrealtionID();
```

Using Server-side JMS Attributes

Overview

When working with JMS servers you get the JMS header information using the JMS server context which is accessed using the `JMS_SERVER_CONTEXT` tag. The JMS client context information is returned as a `JMSServerHeadersType` object. The JMS server context contains all of the JMS header attributes plus an additional boolean attribute called `CommitMessage`.

CommitMessage

`CommitMessage` specifies if a message that is part of a transaction should be commited if an exception is thrown. The default behavior of JMS is to rollback the message and continue to retry a message that is part of a transaction. Setting `CommitMessage` to true before you send the message forces JMS to commit the message regardless of the result of the transmission.

Setting server attributes

As with the JMS header properties on the client-side, the server can only change a few of the values in the JMS header. It can add user-defined properties to the response's JMS header as shown in "Creating optional JMS header properties" on page 354. From the server you can also set a response's delivery mode, priority, and time to live. To set these properties, you use the JMS server context from the server's reply context container. The values are valid only for the active response and are reset each time the servant is invoked.

Example 231 shows the code for setting the JMS header attributes for a response.

Example 231: *Setting a Response's JMS Header Attributes*

```
import com.iona.schemas.transports.jms.context.*;
import com.iona.jbus.ContextConstants;

IONAMessageContext cont = (IONAMessageContext)
  DispatchLocals.getCurrentMessageContext();
```

Example 231:*Setting a Response's JMS Header Attributes*

```
JMSServerHeadersType header = (JMSServerHeadersType)
  cont.getReplyContext(ContextConstants.JMS_SERVER_CONTEXT,
                       true);

header.setJMSDeliveryMode("NON_PERSISTENT");
header.setJMSPriority(new Integer(1));
header.setTimeToLive(new Long(3000));
header.setCommitMessage(Boolean.TRUE);
```

Inspecting server attributes

To inspect the JMS header values of a request message, you get the JMS server context from the server's request context container. Example 230 shows code for checking a request's `JMSRedilvered` flag.

Example 232:*Checking a Request's JMSRedlivered Flag*

```
import com.iona.schemas.transports.jms.context.*;
import com.iona.jbus.ContextConstants;

// Make invocation on proxy

IONAMessageContext cont = (IONAMessageContext)
  DispatchLocals.getCurrentMessageContext();

JMSServerHeadersType header = (JMSServerHeadersType)
  cont.getResponseContext(ContextConstants.JMS_SERVER_CONTEXT,
                          true);

if (header.isJMSRedelivered())
{
  System.out.println("This is a redelivered message.");
}
```

Setting JMS Broker Security Information

Overview

When using a secure JMS broker, your applications will need to register with the JMS broker using a username and password. These are set using the JMS broker connection security property. You need to set this property for both JMS client applications and JMS server applications.

Because the username and password used to connect to the JMS broker must be known before the JMS transport is initialized, you need to set the property in the special configuration context that is made available before Artix registers any user level code with the bus. For more information on using the configuration context see "Using the Configuration Context" on page 312.

Getting the JMS broker connection info

To set the JMS broker connection security information property you use the `ContextConstants` member `JMS_CONNECTION_SECURITY_INFO`. You are returned a `JMSConnectionSecurityInfoType` object that has four methods:

- `setUsername()` sets a `String` representing the username used when connecting to the JMS broker.
- `getUsername()` returns a `String` representing username used when connecting to the JMS broker.
- `setPassword()` sets a `String` representing the password used when connecting to the JMS broker.
- `getPassword()` returns a `String` representing the password used when connecting to the JMS broker.

Example

Example 233 shows how to set the JMS broker connection properties on an Artix JMS client.

Example 233: *Setting the JMS Connection Info*

```
1  ContextRegistry registry = bus.getContextRegistry();

2  QName name = new QName("http://www.iona.com/config_context",
                          "SOAPService");
```

Example 233:*Setting the JMS Connection Info*

```
3  ContextContainer cnt = registry.getConfigurationContext(name,
                                                           "SoapPort",
                                                             true);

4  JMSConnectionSecurityInfoType info =
     (JMSConnectionSecurityInfoType)container.getContext(
                     ContextConstants.JMS_CONNECTION_SECURITY_INFO,
                      true);

5  info.setUsername("george");
   info.setPassword("bosco");
   ...
6  QName servName = new QName("http://buystuff.com", "Register");
   String portName = new String("RegisterPort");
   String wsdlPath = "file:/./resister.wsdl";
   URL wsdlURL = new File(wsdlPath).toURL();
   Register proxy = bus.createClient(wsdlURL, servName,
                                     portName, Register.class);
```

The code in Example 233 does the following:

1. Get the context registry.
2. Create the service's `QName`.
3. Get the configuration context container.
4. Get the client's JMS connection info.
5. Set the username and password.
6. Register the servant.

i18n Attributes

Overview

Artix has two contexts to configure codeset conversion when using the i18n interceptor. One context configures the client and the other configures the server. The i18n interceptor is used when working in an environment where codeset conversion is required, but the transports in use do not support it. It is a message-level interceptor and is invoked just before the transport layer is handed the message.

The i18n interceptor can also be set up using port extensors in your application's contract. For information on setting up the i18n interceptor using port extensors see the chapter on services in Designing Artix Solutions.

Configuring Artix to use the i18n interceptor

Before your application can use the i18n interceptor for code conversion you must configure the Artix bus to load the required plug-ins and add the interceptor to the appropriate message interceptor lists. To configure your application to use the i18n interceptor do the following:

1. If your application includes a service proxy that needs to use codeset conversion, add `"i18n-context:I18nInterceptorFactory"` to the `binding:artix:client_message_interceptor_list` variable for your application.
2. If your application includes a service that needs to use codeset conversion, add `"i18n-context:I18nInterceptorFactory"` to the `binding:artix:server_message_interceptor_list` variable for your application.
3. Add `"i18n_interceptor"` to the list of plug-ins to load in the `orb_plugins` variable for your application.

For more information on configuring Artix see Deploying and Managing Artix Solutions.

Setting up i18n on a client

In a client the only attributes in the i18n context that alter how the i18n interceptor works are the client local codeset and the client outbound codeset in the client's request context. The client inbound codeset defaults to the value of the outbound codeset and the client-side interceptor does not read its value from the context.

To configure a client for codeset conversion using the i18n interceptor do the following:

1. Get the client's message context.
2. Get the i18n client request context.
3. Set the local codeset property.
4. Set the outbound codeset property.

Example 234 shows the code for configuring a client for codeset conversion.

Example 234: *Client i18n Properties*

```
// Java
1 IONAMessageContext messCont =
    (IONAMessageContext)DispatchLocals.getCurrentMessageContext();

2 com.iona.schemas.bus.i18n.context.ClientConfiguration i18nConfig
    = (com.iona.schemas.bus.i18n.context.ClientConfiguration)
    messCont.getRequestContext(
              ContextUtils.I18N_INTERCEPTOR_CLIENT_QNAME, true);

3 i18nConfig.setLocalCodeSet("Latin-1");
4 i18nConfig.setOutboundCodeSet("UTF-16");
```

Setting up i18n on a server

In a server the only attributes in the i18n context that alter how the i18n interceptor works are the server local codeset and the server outbound codeset in the server's reply context. The server-side interceptor does not read the server inbound codeset from the context.

To configure a server for codeset conversion using the i18n interceptor do the following:

1. Get the server's message context.
2. Get the i18n server reply context.
3. Set the local codeset property.
4. Set the outbound codeset property.

Example 235 shows the code for configuring a server for codeset conversion.

Example 235: *Server i18n Properties*

```
// Java
```

Example 235:*Server i18n Properties*

```
1  IONAMessageContext messCont =
     (IONAMessageContext)DispatchLocals.getCurrentMessageContext();

2  com.iona.schemas.bus.i18n.context.ServerConfiguration i18nConfig
     = (com.iona.schemas.bus.i18n.context.ServerConfiguration)
     messCont.getReplyContext(
              ContextUtils.I18N_INTERCEPTOR_CLIENT_QNAME, true);

3  i18nConfig.setLocalCodeSet("UTF-16");
4  i18nConfig.setOutboundCodeSet("LATIN-1");
```

Part II

Advanced Artix Programming

In this part

This part contains the following chapters:

CHAPTER 13

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

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 randomly or using a round-robin algorithm.

Figure 7 gives a general overview of the locator architecture.

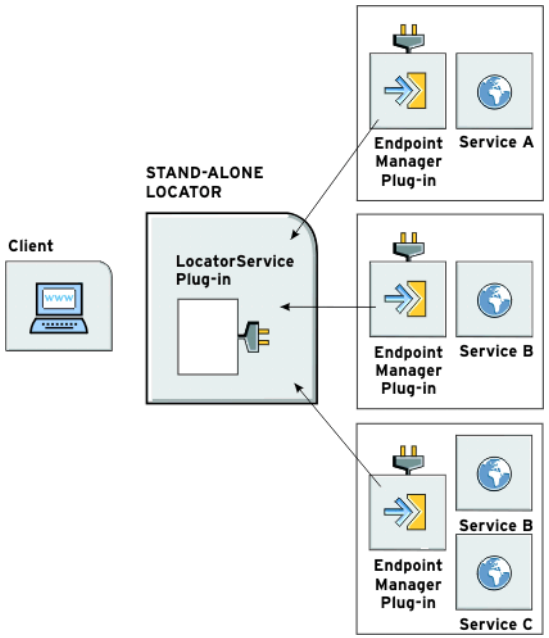

Figure 7: *Artix Locator Overview*

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 7). 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/advanced/locator` demo.
- *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 plug-in 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's default behavior is to use a round-robin algorithm to pick one of the endpoints. Hence, the locator effectively *load balances* a service over all of its associated endpoints.

In addition to using a round-robin load balancing algorithm, the locator also supports the randomly selecting one of the registered endpoints from the list of registered services. When the locator is configured to use persistent registration data, it automatically switches to random load balancing. You can also configure the locator to always use random load balancing by setting the `plugins:locator:selection_method` configuration variable to `random`.

For example, Figure 7 on page 370 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.

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 236.

Example 236: *Server Configuration Scope for Using the Locator*

```
# Artix Configuration File (artix.cfg)
...
located_server
{
  orb_plugins = ["xmlfile_log_stream", "soap", "at_http",
                 "locator_endpoint"];
  bus:initial_contract:url:locator="../wsdl/locator.wsdl";
};
```

It must also specify where the locator instance's contract can be found. This information is specified using the `bus:initial_contract_url:locator` configuration variable. The default location of the contract is `artix/Version/wsdl/locator.wsdl`. The default contract sets up the locator to start on a system determined port.

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 8 shows an outline of how a client connects to a server in this way.

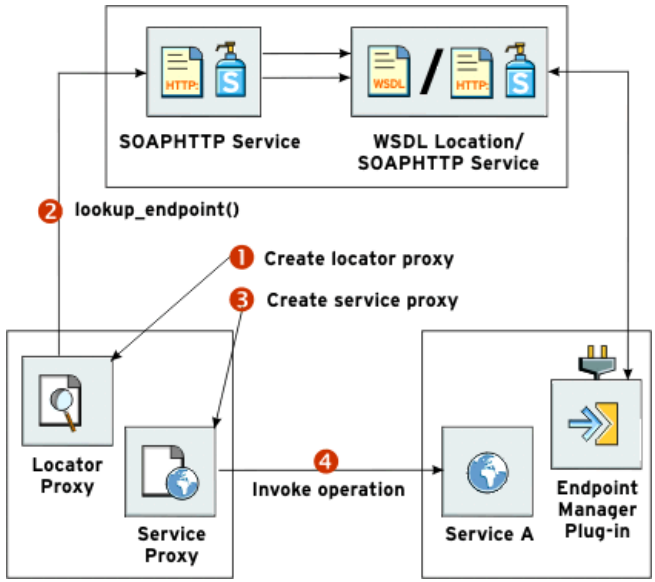

Figure 8: *Steps to Read a Reference from the Locator*

Programming steps

The main programming steps needed to read a reference from the locator, as shown in Figure 8, are as follows:

1. Construct a locator service proxy.
2. Use the locator proxy to invoke `lookup_reference()`.

3. Use the reference returned from `lookup_reference()` to construct a proxy to the service.
4. Invoke an operation using the created service proxy.

Example

Example 237 shows an example of the code for an Artix client that retrieves a reference to a `SimpleService` service from the Artix locator.

Example 237: *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);

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");
```

Example 237:*Reading a Reference from the Locator Service*

```
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 237 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 contract.
3. This line constructs a QName, `lookup_name`, that identifies the `SOAPHTTPService` service from the `SimpleService` contract.
4. This port name refers to the `<port name="LocatorServicePort" ...>` tag in the locator contract.
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 23.
6. The `lookup_endpoint()` operation is invoked on the locator to find an endpoint of `SOAPHTTPService` type.

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 14

Using Sessions in Artix

The Artix Session Manager helps you manage service resources.

In this chapter

This chapter discusses the following topics:

Introduction to Session Management in Artix

Overview

The Artix session manager is a group of ART plug-ins that work together to provide you control over the number of concurrent clients accessing a group of services and how long each client can use the services in the group before having to check back with the session manager. The two main session manager plug-ins are:

Session Manager Service Plug-in (`session_manager_service`) is the central service plug-in. It accepts and tracks service registration, hands out session to clients, and accepts or denies session renewal.

Session Manager Endpoint Plug-in (`session_endpoint_manager`) is the portion of the session manager that resides in a registered service. It registers its location with the service plug-in and accepts or rejects client requests based on the validity of their session headers.

The session manager also has a pluggable policy callback mechanism that allows you to implement your own session management policies. Artix session manager includes a simple policy callback plug-in, `sm_simple_policy`, that provides control over the allowable duration for a session and the maximum number of concurrent sessions allowed for each group.

How do the plug-ins interact?

Figure 9 shows a diagram of how the session manager plug-ins are deployed in an Artix System. As you can see the session manager service plug-in and the policy callback plug-in are both deployed into the same process. While in this example, they are deployed into a standalone service, they can be deployed in any Artix process. The session manager service plug-in and the policy plug-in interact to ensure that the session manager does not hand out sessions that violate the policies established by the policy plug-in.

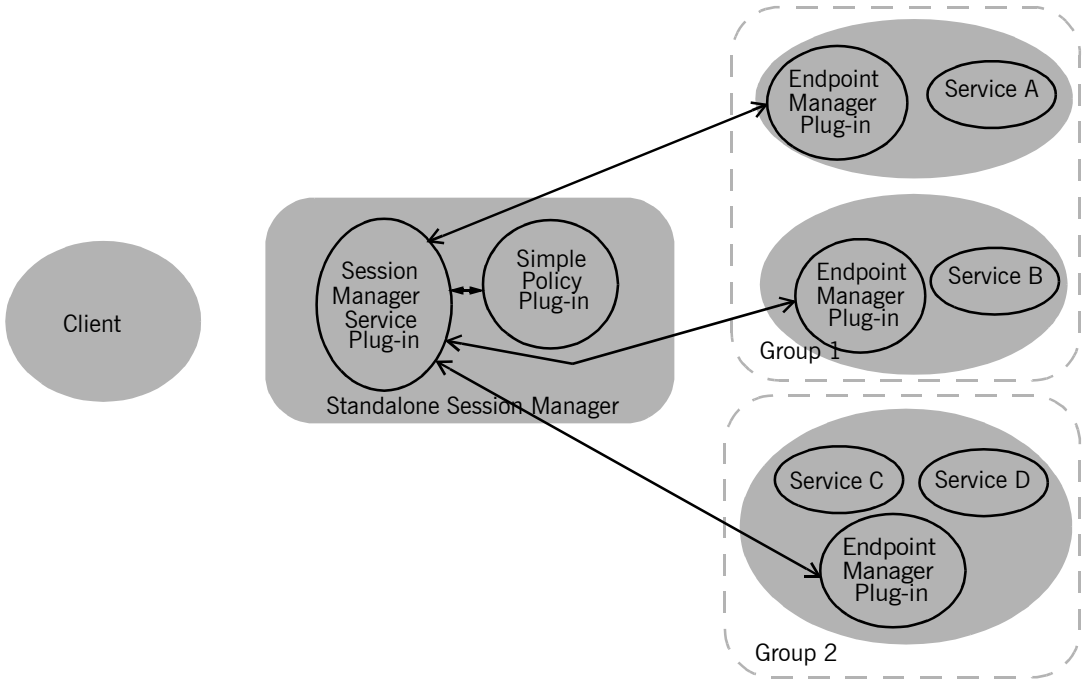

Figure 9: *The Session Manager Plug-ins*

The endpoint manager plug-ins are deployed into the server processes which contain session managed services. A process can host two services, like *Service C* and *Service D* in Figure 9, but the process will have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the session manager service plug-in to report on

endpoint health, to receive information on new sessions that have been granted to the managed services, and to check on the health of the session manager service.

What are sessions?

The session manager controls access to services by handing out *sessions* to clients who request access to the services. A session is a pass that provides access to the services in a specific group for a specific time.

For example if a client application wants to use the services in the `SM_Demo` group, it would ask the session manager for a session with the `SM_Demo` group. The session manager would then check and see if the `SM_Demo` group had an available session, and if so it would return a session id and the list of `SM_Demo` service references to the client. The session manager would then notify the endpoint managers in the `SM_Demo` group that a new session had been issued, the new session's id, and the duration for which the session is valid. When the client then makes requests on the services in the `SM_Demo` group, it must include the session information as part of the request. The endpoint manager for the services then check the session information to ensure it is valid. If it is, the request is accepted. If it is not, the request is rejected.

If the client wants to continue using the `SM_Demo` services beyond the duration of its lease, the client will have to ask the session manager to renew its session before the session expires. Once a client's session has expired, it will have to request a new one.

What are groups?

The Artix session manager does not pass out sessions for each individual service that is registered with it. Instead, services are registered as part of a *group*, and sessions are handed out for the group. A group is a collection of services that are managed as one unit by the session manager. While the session manager does not specify that the services in a group be related, it is recommended that the endpoints have some relationship.

A service's group affiliation is controlled by the configuration scope under which it is run. To change a service's group, you edit the value for `plugins:session_endpoint_manager:default_group` in the process' configuration scope. For more information on Artix configuration see Deploying and Managing Artix Solutions.

Registering a Server with the Session Manager

Overview

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

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

Configuring the server

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

- `soap`
- `at_http`
- `session_endpoint_manager`

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

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

bus:initial_contract:url:sessionmanager points to the contract describing the contact information for the session manager that will be managing the services.

bus:initial_contract:url:sessionendpointmanager points to the contract describing the contact information for the endpoint manager for this server. This enables the session manager to contact the service to with updated state information.

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

Example 238 shows the configuration scope of a server that hosts services managed by the session manager.

Example 238: *Server Configuration Scope*

```
demos {
    session_management {
        server {
            orb_plugins = ["xmlfile_log_stream", "session_endpoint_manager"];

            # This is the WSDL File that the Session Endpoint Manager used to contact the
            # Session Manager Service.
            bus:initial_contract:url:sessionmanager = "../../etc/session-manager.wsdl";
            bus:initial_contract:url:sessionendpointmanager = "../../session-manager.wsdl";
            plugins:session_endpoint_manager:default_group = "SM_Demo";
        };
    };
};
```

A server loaded into the `demos.session_management.server` configuration scope will be managed by the session manager at the location specified in `session-manager.wsdl` and by default all services instantiated by the server will belong to the session manager group `SM_Demo`.

For more information on Artix configuration see Deploying and Managing Artix Solutions.

In the server's configuration scope specify the endpoint manager plug-in to read the correct Artix contract for the endpoint manager by setting `bus:initial_contract:url:sessionendpointmanager` to point to the copy of `session-manager.wsdl` containing the address for this instance of the endpoint manager. It is recommended that you use the default `session-manager.wsdl` shipped with Artix and not specify dedicated ports for your endpoint manager.

Registration

Once a properly configured server starts up, it automatically registers with the session manager specified by the contract pointed to by `bus:initial_contract:url:sessionmanager`.

Working with Sessions

Overview

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

Demonstration code

The examples in this section are based on the demonstration code located in the following directory:

ArtixInstallDir`/artix/`*Version*`/demos/advanced/session_management`

Implementing a session client

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

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

Registering the session manager's type factory

Artix uses the context mechanism to pass session information between the session manager, clients, and services. Therefore you must register the session manager's type factory with the bus before making any calls on the session manager or session managed services.

Example 239 shows the code for registering the session manager's type factory.

Example 239: *Registering the Session Manager's Type Factory*

```
// bus obtained earlier
bus.registerTypeFactory(new
        com.iona.ws.sessionmanager.SessionManagerTypeFactory());
```

Instantiating a session manager proxy

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

Example 240 shows how to instantiate a session manager proxy.

Example 240: *Instantiating a Session Manager Proxy*

```
QName name = new QName("http://ws.iona.com/sessionmanager",
                       "SessionManagerService");
QName portName = new QName("", "SessionManagerPort");

URL wsdlLocation = null;
try
{
  wsdlLocation = new URL(wsdlPath);
}
catch (java.net.MalformedURLException ex)
{
  wsdlLocation = new File(wsdlPath).toURL();
}

ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(wsdlLocation, name);
SessionManager sessionMgr =
   (SessionManager)service.getPort(portName,
   SessionManager.class);
```

For more information on instantiating Artix proxies, see the "Proxy Creation" on page 40.

Start a session

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

```
SessionInfo begin_session(String endpoint_group,
                          BigInteger preferred_renew_timeout);
```

The `begin_session()` function takes the following input parameters:

- `endpoint_group`—the endpoint group name, which corresponds to the default group name set in the server's configuration scope as described in "Configuring the server" on page 383.
- `preferred_renew_timeout`—the preferred session duration in seconds. If the specified duration is less than the value specified by the session manager's `min_session_timeout` configuration setting, it will be set to the configured minimum value. If the specified duration is higher than the value specified by the session manager's `max_session_timeout` configuration setting, it will be set the configured max value.

And returns the following:

- `SessionInfo`—a sequence complex type that contains the session id, `session_id`, and the actual assigned session duration, `renew_timeout`.

Example 241 shows the client code to begin a session for `SM_Demo`.

Example 241: *Beginning a Session*

```
SessionInfo sessionInfo = null;
String _endpoint_group = "SM_Demo";
BigInteger _preferred_renew_timeout = new
   Java.math.BigInteger("20");
sessionInfo = sessionMgr.begin_session(_endpoint_group,
                                       _preferred_renew_timeout);
```

Get a list of endpoints in the group

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

```
EndpointList get_all_endpoints(SessionId session_id);
```

The `get_all_endpoints()` function takes the following input parameter:

- `session_id`—the session ID for which you are requesting services (obtained in the previous step).

And returns the following output:

- `endpoints`—the list of services available. If the group has no services, the list will be empty.

Example 242 shows how to get the list of services for a group.

Example 242: *Retrieving the List of Services in a Group*

```
EndpointList endptList = null;

endptList =
   sessionMgr.get_all_endpoints(sessionInfo.getSession_id());
System.out.println("Obtained endpoints...");
System.out.println(endpointList.toString());
```

Create a proxy for the requested service

The client can use any of the services returned by `get_all_endpoints()` to instantiate a service proxy.

Because the session manager simply returns the services in the order the services registered with the session manager, the clients are responsible for circulating through the list or else they will all make requests on only one service in the group. Also, because the session manager does not force all members of a group to implement the same interface, you might need to have your clients check each service to see if it implements the correct interface by checking the reference's service name as shown in Example 243.

Example 243: *Checking the Service Reference for its Interface*

```
Reference[] references = endpointList.getEndpoint();
if (references[0].get_service_name() ==
    QName("", "QajaqService", "http://qajaqs.com"))
  {
  // instantiate a QajaqService using endpoint
  }
else
  {
  // do something else
  }
```

Example 244 shows the client code for creating a `GreeterClient` proxy from an endpoint reference.

Example 244: *Instantiate a Proxy Server*

```
Reference[] references = endpointList.getEndpoint();
Greeter greeter = (Greeter)bus.createClient(references[0],
                                            Greeter.class);
```

Create a session header

Services that are being managed by the session manager will only accept requests that include a valid session header. Example 245 shows how to send the session ID in a header by initializing the `sessionIDContext` header context. For more details about the context API used in this example, see "Using Message Contexts" on page 267.

Example 245: *Initialize the sessionIDContext Header Context*

```
ContextRegistry registry = bus.getContextRegistry();

QName principalCtxName = new QName("", "SessionId");
QName principalCtxType = new
   QName("http://ws.iona.com/sessionmanager", "SessionId");
QName principalMessageName = new
   QName("http://ws.iona.com/sessionmanager", "", "");
String principalPartName = "id";

registry.registerContext(principalCtxName,
                         principalCtxType,
                         principalMessageName,
                         principalPartName);

SessionId sessionId = sessionInfo.getSession_id();

IonaMessageContext contextImpl =
   (IonaMessageContext)registry.getCurrent();
contextImpl.setRequestContext(principalCtxName, sessionId);
```

Make requests on service proxy

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

Renewing a session

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

```
BigInteger renew_session(SessionInfo session_info);
```

The `renew_session()` function takes the following input parameter:

- `session_info`—a sequence complex type that contains the session id, `session_id`, and the preferred session duration, `renew_timeout`.

And the following output parameter

- `BigInteger`—the actual assigned session duration, in seconds.

If the renewal is unsuccessful, an exception is raised.

End the session

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

```
void end_session(SessionId);
```

Example 246 shows how to end a session.

Example 246: *Ending a Session*

```
sessionMgr.end_session(sessionId);
```

CHAPTER 15

Using Persistent Datastores

Artix provides a persistence mechanism, built on top of Berkeley DB, which you can use to persist data when using Artix. With this mechanism, you can make your services highly available.

In this chapter

This chapter discusses the following topics:

Introduction to Artix Persistent Datastores

Overview

In many enterprise services it is imperative that data does not get lost when a service goes down. There are also many instances where an enterprise service must always be available. To address these usecases, Artix has an integrated persistence mechanism. This mechanism, which is built using Berkeley DB, provides a Java API for storing data in persistent datastores as shown in Figure 10.

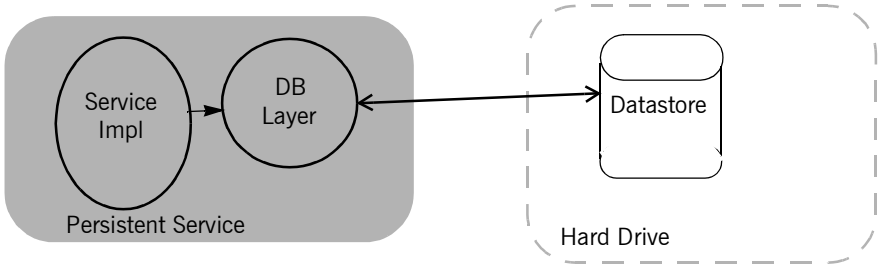

Figure 10: *The Artix Persistence Mechanism*

In addition, the persistence mechanism provides the backbone for creating highly available services. Services that are implemented using persistent datastores can be configured and deployed in a highly available cluster as shown in Figure 11. The Berkeley DB layer will seamlessly set up a master/slave relationship between members of the cluster to ensure that the

service remains available and the slaves have the latest data from the master. To use write-forwarding from the slaves, you need to do a bit of extra coding.

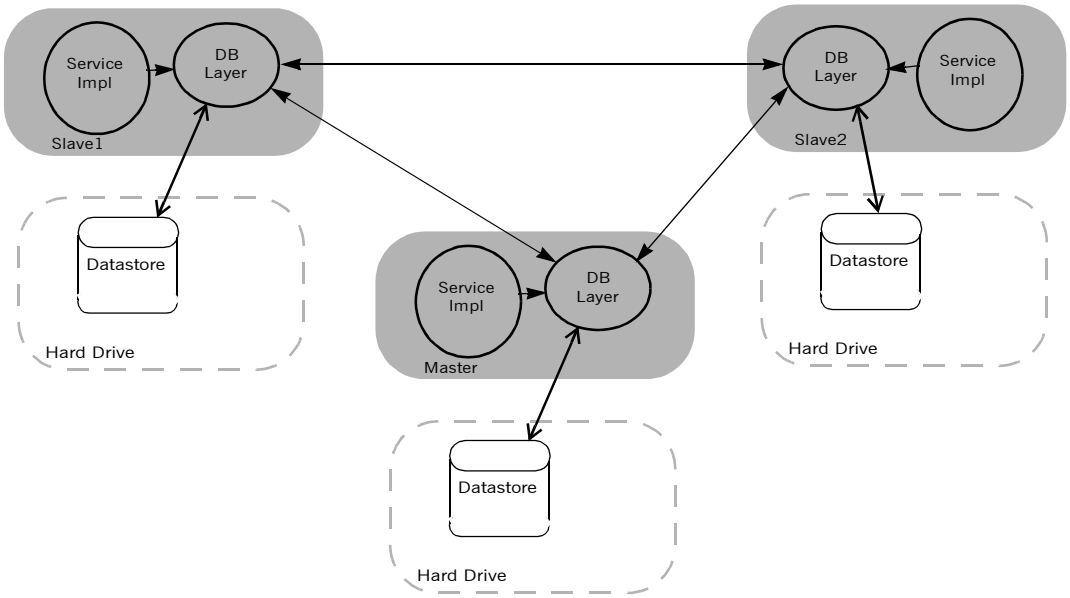

Figure 11: *Artix Service Cluster*

For more information on deploying your service as a highly available cluster see Deploying and Managing Artix Applications.

How Artix datastores are structured

Artix persistent datastores are hash tables stored in a Berkeley DB database. The hash table stores pairs of items as shown in Figure 12.The first item is a *key* and the second item is the *data*. Both the key, which is used to locate

entries in the datastore, and the data can be any Java object. The objects can either be stored as serialized data, or, if they are generated by Artix, as XML data.

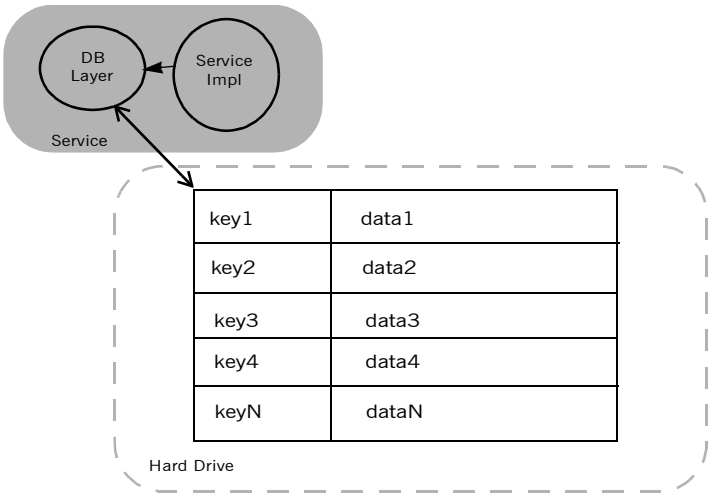

Figure 12: *Artix Persistent Datastores*

Developing a service with persistent datastores

Developing a service that uses Artix based persistent datastores is a simple process. To create a persistent datastore and work with the data it contains you will need to do the following:

1. Create a database manager object.
2. Create one or more persistent datastores using the provided templates.
3. Use the persistent datastore object to add or remove data from the persistent datastore.
4. Close the persistent datastore.
5. Close the database manager.

The APIs deal exclusively with creating datastores and manipulating the data stored in them. The underlying Berkeley DB layer automatically creates a new database instance for the service's datastores and initializes all of the database connections. The Berkeley DB layer's behavior can be configured

to specify the location of the database and the name of the Berkeley DB's environment file. By default the database and environment files are created in the directory from which the service is started.

Packages

To use persistent datastores in an Artix application you will need to import the following packages:

com.iona.jbus.db contains the classes for configuring the database layer and handling exceptions thrown by the database layer.

com.iona.jbus.db.collections contains the template classes from which you instantiate instances of Artix datastores.

Types of Persistent datastores

Artix provides two different types of persistent datastores. You can choose persistent datastores the are implementations of `java.util.Map` or you can choose datastores that are implementations of `java.util.List`. Both types of datastore use the database layer to automatically persist data.

The key difference between the two types of datastores is how they handle the keys in the hash table. Using persistent maps, you get to specify the key values. When you use persistent lists, the key values of the hash table are handled by the database layer. They are always a sequential series of integers.

Persistent map templates

There are four templates for using persistent maps:

- `PersistentMap` is the base class for all persistent maps. It allows you to store data in any format for which you have a data handler. The most common use is to store both key values and data values as XML.
- `SerialPersistentMap` allows both the key values and the data values to be any serializable Java object.
- `StringSerialPersistentMap` allows key valuess to be java `String` objects and the data values to be any serializable Java object.
- `StringXMLPersistentMap` allows key values to be Java `String` objects and the data values to be an Artix generated Java object that will be stored as XML.

Persistent list templates

There are two persistent list templates:

- `PersistentList` is the base class for all persistent lists. It allows you to store data in any format for which you have a data handler. The most common use is to store data values as XML.
- `SerialPersistentList` allows you to store any serializeable Java object.

Creating a Persistent Datastore

Overview

Artix persistent datastores are instances of one of the persistent datastore templates listed in "Types of Persistent datastores" on page 395. The first step in creating a persistent datastore is to consider what data is going to be stored in the datastore and in what format you want it stored. For example, if you are storing a complex type defined in one of your contracts, you do not care what the key values are, and want to be able to access the data as XML that can be read by an Artix C++ service, then you may want to make your datastore an instance of `PersistentList`. If you want the data to be keyed using strings and want it accessible to a Java application, you may want to make your datastore an instance of `StringSerialPersistentMap`.

In this section

This section contains subsections discussing the following topics:

Procedure

To create a persistent datastore you need to do four things:

1. Determine what type of datastore you want to create.
2. Instantiate a `DatabaseManager` object to hold the database configuration.
3. If the datastore you want to create stores Artix generated datatypes as XML, create an `XMLDataHandler` for each type.
4. Instantiate an instance of the persistent datastore template for the type of datastore is most appropriate for your application.

Instantiating a DatabaseManager

To instantiate an instance of a `DatabaseManager` object for your service you pass an instance of the active bus into its constructor as shown in Example 247.

Example 247: *Instantiating a DatabaseManager*

```
import com.iona.jbus.*;
import com.iona.jbus.db.*;

Bus bus = Bus.init(args);
DatabaseManager mgr = new DatabaseManager(bus);
```

When the database manager is instantiated, Artix initiates the database layer. The database manager is used when creating persistent datastores. It also provides a method for releasing database locks when using iterators created by datastores created with it.

Closing the DatabaseManager

When your application is done accessing persistent data, you need to invoke the database manager's `close()` method. This releases any resources used in maintaining the connection to the underlying database and ensures that it is left in a stable state.

WARNING: This must be done before the server is shutdown.

Creating an XMLDataHandler

An `XMLDataHandler` object provides the database layer with the information needed to convert an object into an XML document. To create an `XMLDataHandler` object for an Artix generated class you need the following things:

- The `QName` of the root element of the XML representation of the data in the datastore.
- The `QName` of the XMLSchema type that defines the class.
- The `Class` object for the class.
- The location of the contract in which the type is defined.

Example 248 shows an example of creating an `XMLDataHandler` object for the `widgetOrderInfo` type defined in Example 79 on page 115.

Example 248:*Creating an XMLDataHandler*

```
QName typeName = new QName("http://widgets.com/widgetTypes",
                           "widgetOrderInfo");
String wsdlPath = "file:/../widgets.wsdl";

XMLDataHandler handler = new XMLDataHandler(null, typeName,
                                            WidgetOrderInfo.class,
                                              wsdlPath);
```

Creating Persistent Maps

Overview

All of the persistent datastore templates that implement `java.util.Map` extend from the superclass `PersistentMap`. They also share two instantiation parameters:

- *`id`* - specifies the name of the datastore. It can be any string value. If a datastore matching the id already exists, the database layer will connect to that datastore. If the datastore does not exist, the database layer will create a new datastore.
- *`manager`* - specifies the database manager that provides the connection to the database layer.

Each of the templates that extend `PersistentMap` have additional parameters that are required to instantiate them. The following blocks describe each.

Creating a generic PersistentMap

To create a generic `PersistentMap` you need to pass in the id of your map, the database manager, and two `Datahandler` objects. The first is for the key value and the second one is for the data value. If you chose not to use the supplied `XMLDataHandler` objects you can create your own custom data handlers by extending the `com.iona.jbus.db.collections.DataHandler` interface.

The most common use for a generic persistent map is to store Artix generated objects that are defined in XMLSchema as XML. This is done by passing in an `XMLDataHandler` for both the key and the data. When an object is placed into the map both the key and the data are converted into XML based on their schema definitions. The XML representations are then written into the persistent store.

Note: If you want to share a persistent datastore between a Java service and a C++ service, you will need to use a persistent map that stores data as XML.

When using this type of persistent map both your key and data must be Artix generated objects and the service must have access to the XMLSchema definitions of the types. Objects not defined in an accessible XMLSchema will cause an exception to be thrown.

Example 249 shows how to instantiate a `PersistentMap` that stores objects as XML. The id of the created datastore is `widget_table`.

Example 249: *Instantiating a PersistentMap for storing XML*

```
import com.iona.jbus.db.collections.*;

String wsdlPath = "file:/../widgets.wsdl";

QName keyName = new QName("http://widgets.com/widgetTypes", "orderID");
QName dataName = new QName("http://widgets.com/widgetTypes", "widgetOrderInfo");

XMLDataHandler keyHandler = new XMLDataHandler(null, keyName, OrderID.class, wsdlPath);
XMLDataHandler dataHandler = new XMLDataHandler(null, dataName, WidgetOrderInfo.class, wsdlPath);

// DatabaseManager mgr obtained earlier
PersistentMap widgetMap = new PersistentMap("widget_table", mgr, keyHandler, dataHander);
```

Creating a SerialPersistentMap

A `SerialPersistentMap` is the most flexible of the persistent datastore templates. It allows you to use any serializable Java object for both the key and data in your map. To create an instance of a `SerialPersistentMap`, you pass in the id of the database you wish to create, the database manager for the datastore, and the `Class` objects for both the key and the data to be stored in the map.

The only restriction on the type of data that can be stored in a `SerialPersistentMap` is that the objects must be serializable. All native Java objects are serializable. However, Java atomic types, such as `long`, are not serializable. Also, object generated by Artix are not, be default serializable. To make Artix generated objects serializable use the `-ser` flag when using `wsdltojava`.

Example 250 shows how to instantiate a `SerialPersistentMap` that uses `Integer` objects as keys and `Inet6Address` objects as data. The id of the created datastore is `host_ipv6_table`.

Example 250: *Instantiating a SerialPersistentMap*

```
import com.iona.jbus.db.collections.*;

// DatabaseManager mgr obtained earlier
SerialPersistentMap ipMap = new SerialPersistentMap("host_ipv6_table", mgr, Integer.class,
   Inet6Address.class);
```

Creating a StringSerialPersistentMap

A `StringSerialPersistentMap` allows you to store any serializable Java object as data but it requires that the key values be strings. To create an instance of a `StringSerialPersistentMap`, you pass in the id of the database you wish to create, the database manager for the datastore, and the `Class` objects for the data to be stored in the map.

Example 251 shows how to instantiate a `StringSerialPersistentMap` that stores `Float` objects as data. The id of the created datastore is `float_table`.

Example 251: *Instantiating a StringSerialPersistentMap*

```
import com.iona.jbus.db.collections.*;

// DatabaseManager mgr obtained earlier
StringSerialPersistentMap floatMap = new StringSerialPersistentMap("float_table", mgr,
   Float.class);
```

Creating a StringXMLPersistentMap

A `StringXMLPersistentMap` uses strings as the key values and the XML representation of an Artix generated object that is defined in XMLSchema as the data. When an object is placed into the map the data is converted into XML based on their schema definitions. The XML representation is then written into the persistent store.

When using this type of map the data must be an Artix generated object and the service must have access to the XMLSchema definitions of the type the object represents. Objects not defined in an accessible XMLSchema will cause an exception to be thrown.

To create a `StringXMLPersistentMap` you need to pass in the id of your map, the database manager, and an `XMLDatahandler` object for the data value.

Example 252 shows how to instantiate a `StringXMLPersistentMap`. The id of the created datastore is `widget_table`.

Example 252: *Instantiating a StringXMLPersistentMap*

```
import com.iona.jbus.db.collections.*;

String wsdlPath = "file:/../widgets.wsdl";

QName dataName = new QName("http://widgets.com/widgetTypes", "widgetOrderInfo");
```

Example 252:*Instantiating a StringXMLPersistentMap*

```
XMLDataHandler dataHandler = new XMLDataHandler(null, dataName, WidgetOrderInfo.class, wsdlPath);

// DatabaseManager mgr obtained earlier
StringXMLPersistentMap widgetMap = new StringXMLPersistentMap("widget_table", mgr, dataHandler);
```

Creating Persistent Lists

Overview

The two persistent datastore templates that implement `java.util.List` extend from the superclass `PersistentList`. They also share two instantiation parameters:

- *`id`* - specifies the name of the datastore. It can be any string value. If a datastore matching the id already exists, the database layer will connect to that datastore. If the datastore does not exist, the database layer will create a new datastore.
- *`manager`* - specifies the database manager that provides the connection to the database layer.

Each of the templates that extend `PersistentList` have additional parameters that are required to instantiate them. The following blocks describe each.

Creating a generic PersistentList

To create a generic `PersistentList` you need to pass in the id of your list, the database manager, and a `Datahandler` object for the data value. The most common use for a generic persistent list is to store Artix generated objects that are defined in XMLSchema as XML. This is done by passing in an `XMLDataHandler` for the data elements data handler. When an object is placed into the list it is converted into XML based on its schema definition. The XML representations are then written into the persistent store.

Note: If you want to share a persistent datastore between a Java service and a C++ service, you will need to use a persistent list that stores data as XML.

If you chose not to use the supplied `XMLDataHandler` object you can create your own custom data handler by extending the `com.iona.jbus.db.collections.DataHandler` interface.

When using this type of persistent list both your data must be Artix generated objects and the service must have access to the XMLSchema definitions of the type. Objects not defined in an accessible XMLSchema will cause an exception to be thrown.

Example 253 shows how to instantiate a `PersistentList` that stores objects as XML. The id of the created datastore is `widget_list`.

Example 253: *Instantiating a PersistentList for storing XML*

```
import com.iona.jbus.db.collections.*;

String wsdlPath = "file:/../widgets.wsdl";

QName keyName = new QName("http://widgets.com/widgetTypes", "orderID");
QName dataName = new QName("http://widgets.com/widgetTypes", "widgetOrderInfo");

XMLDataHandler dataHandler = new XMLDataHandler(null, dataName, WidgetOrderInfo.class, wsdlPath);

// DatabaseManager mgr obtained earlier
PersistentList widgetList = new PersistentList("widget_table", mgr, dataHander);
```

Creating a SerialPersistentList

A `SerialPersistentList` allows you to store any serializable Java object. To create an instance of a `SerialPersistentList`, you pass in the id of the database you wish to create, the database manager for the datastore, and the `Class` objects for the data to be stored in the list.

The only restriction on the type of data that can be stored in a `SerialPersistentList` is that the objects must be serializable. All native Java objects are serializable. However, Java atomic types, such as `long`, are not serializable. Also, object generated by Artix are not, be default serializable. To make Artix generated objects serializable use the `-ser` flag when using `wsdltojava`.

Example 254 shows how to instantiate a `SerialPersistentList` that stores `Float` objects as data. The id of the created datastore is `float_list`.

Example 254: *Instantiating a SerialPersistentList*

```
import com.iona.jbus.db.collections.*;

// DatabaseManager mgr obtained earlier
SerialPersistentList floatList = new SerialPersistentList("float_table", mgr, Float.class);
```

Working with Data in a Persistent Datastore

Overview

Artix persistent datastores are implemented using the standard Java interfaces `java.util.Map` and `java.util.List`. The Artix implementations are built on top of Berkeley DB to provide persistence so they have a few Artix specific behaviors. They implement all of the defined methods for both interfaces. In addition, they have a method for closing the datastore when the application is finished with it.

In this section

This section discusses the following topics:

Using Persistent Maps

Overview

Artix persistent maps implement `java.util.Map` using Berkeley DB to provide persistence. To manipulate the data in a persistent map you use the standard methods defined for a `Map` object. However, because the maps are persistent there are a things to consider when using them:

- `Iterator` objects are implemented using Berkeley DB cursors that aquires a read lock on the datastore. This lock is not realeased until the `Iterator` object is closed by the database manager.
- When your application is finished working with a persistent map it must close the map or the database layer may leave the data in an unusable state.

Adding data to a map

Maps have two methods for inserting data. The one most likely to be used is `put()`. `put()` takes two objects as parameters:

- The first object is the key.
- The second object is the data.

When using `SerialPersistentMap` maps you must be sure that both the key and the data objects are of the class you specified when creating the map. When using `StringSerialPersistentMap` maps, you must ensure that the key is a String object and that the data is of the class you specified when creating the map. The XML style persistent maps do not have this restriction because the objects are converted to XML representations.

Example 255 shows the code for adding an entry to a `StringSerialPersistentMap` using `put()`.

Example 255: *Putting an Element in a Persistent Map*

```
import com.iona.jbus.db.collections.*;

// DatabaseManager mgr obtained earlier
StringSerialPersistentMap floatMap = new
   StringSerialPersistentMap("float_table", mgr, Float.class);

Float data = new Float(0.314);
floatMap.put("first", data);
```

The other way to add data to a persistent map is to use the `putAll()` method. `putAll()` takes a `Map` object as a parameter and copies all of the values from the map parameter into the current map. If any values in the current map have the same key as a value in the map being copied, the copied values overwrite them.

Removing data from a map

You remove entries from a persistent map using the `remove()` method. `remove()` takes a key value and returns the data value associated with the key. `remove()` deletes the data value associated with the key from the map.

When using persistent maps that use serialized objects as key values, you must be sure to specify the proper class of object for the key. When using persistent maps that use `String` objects as keys, you must ensure that the value used in a `String` object.

Example 256 shows code for removing an object from a map.

Example 256:*Removing an Element from a Persistent Map*

```
floatMap.remove("first");
```

In addition to using `remove()` to delete a single entry from a persistent map, you can also clear all of the entries in a persistent map by invoking its `clear()` method.

Getting an entry from a map

To retrieve an entry from a persistent map you can use the `get()` method. `get()` takes a kay value as a parameter and returns the data value associated with the key. If the key does not exist in the map `get()` returns null.

When using persistent maps that use serialized objects as key values, you must be sure to specify the proper class of object for the key. When using persistent maps that use `String` objects as keys, you must ensure that the value used in a `String` object.

Example 257 shows code for getting an object from a map.

Example 257:*Getting an Element from a Persistent Map*

```
floatMap.get("first");
```

Searching through the map

If you wish to search through all of the data values in a persistent map you will need to use one the two methods that return the data values in a form that provides access to an `Iterator` object:

- `entrySet()` returns the values stored in the map as a `java.util.Set` object.
- `values()` returns the values stored in the map as a `java.util.Collection` object.

Both the `Set` object and the `Collection` object support the `iterator()` method. `iterator()` returns an `Iterator` object that can be used to iterate through the values in the map. Any changes made to values using either the `Set` object, the `Collection` object, or the `Iterator` object are reflected in the values stored in the original persistent map.

The returned `Iterator` object is implemented using Berkeley DB cursors. When the `Iterator` object is created the database layer creates a read lock on the underlying datastore. This read lock is held until the `Iterator` object is closed by the database manager using the database manager's static `closeIterator()` method. `closeIterator()` takes the `Iterator` object to be closed as a parameter.

Example 258 shows code for iterating through a map.

Example 258: *Iterating through a Persistent Map*

```
Iterator iter = floatMap.entrySet().iterator();

while (iter.hasNext())
{
  Map.entry entry = (Map.Entry)iter.next();
  System.out.println(entry.getKey() + ' ' + entry.getValue());
}

DatabaseManager.closeIterator(iter);
```

Closing a persistent map

When you are finished working with a persistent map, your application needs to invoke the persistent map's `close()` method. `close()` informs the database layer to release any resources used to maintain the connection to the physical representation of the datastore and flushes any buffered writes to the physical disk.

Example 259 shows code for closing a persistent map.

Example 259:*Closing a Persistent Map*

```
floatMap.close();
```

Other operations

Artix persistent maps implement all of the methods of the `java.util.Map` interface. These methods provide means for querying the list to see if it contains a specific key values or specific data values. They also provide a means for seeing if the map has any data stored in it. For a full list of all the methods available see the Java 1.4.2 API documentation for `java.util.Map` (http://java.sun.com/j2se/1.4.2/docs/api/java/util/Map.html).

Note: Artix persistent maps throw an unsupported exception when invoking the `size()` method.

Using Persistent Lists

Overview

Artix persistent lists implement `java.util.List` using Berkeley DB to provide persistence. To manipulate the data in a persistent list you use the standard methods defined for a `List` object. However, because the lists are persistent there are a things to consider when using them:

- `Iterator` objects are implemented using Berkeley DB cursors that aquires a read lock on the datastore. This lock is not realeased until the `Iterator` object is closed by the database manager.
- When your application is finished working with a persistent list it must close the list or the database layer may leave the data in an unusable state.

Adding data to a list

Lists have four methods that can be used to add data:

- `add(Object obj)` adds the specified to the end of the list.
- `add(int index, Object obj)` adds the specified object to the specified position in the list and shifts all existing elements that fall after the new object are forward one element.
- `addAll(Collection col)` adds the objects stored in the specified `Collection` object to the end of the list.
- `addAll(int index, Collection col)` adds the object stored in the specified in the `Collection` object to the list starting at the specifed position. The elements that fall after the newly inserted objects are are shifted forward in the list.

When using a `SerialPersistentList` you need to ensure that all of the objects being added to the list are of the class specified when the list was created.

Example 260 shows an example of adding an element to the end of a persistent list.

Example 260: *Adding an Element to a Persistent List*

```
import com.iona.jbus.db.collections.*;

// DatabaseManager mgr obtained earlier
SerialPersistentList floatList = new
   SerialPersistentList("float_table", mgr, Float.class);

Float data = new Float(0.314);
floatList.add(data);
```

Removing data from a list

Lists have four methods for removing data:

- `clear()` deletes all of the entries from the list.
- `remove(int index)` removes the entry specified by the index. The elements that come after the removed element are shifted back by one.
- `remove(Object obj)` removes the specified object from the list. The elements that come after the removed element are shifted back by one.
- `remove(Collection col)` removes all of the elements in the collection from the list. The remaining elements are adjusted to remove any gaps.

Example 261 shows an example of removing an element from a persistent list.

Example 261: *Removing an Element from a Persistent List*

```
floatList.remove(3);
```

Getting an element from a list

To retrieve a single element from a persistent list you use the `get()` method. `get()` takes an integer value and returns the entry stored at the specified position in the list.

Example 261 shows an example of getting an element from a persistent list.

Example 262: *Getting an Element from a Persistent List*

```
floatList.get(3);
```

Searching through the elements of a list

If you wish to search through all of the elements in a persistent list you will need to use one the three methods that return an `Iterator` object:

- `iterator()` returns an `Iterator` object to access the entries in their proper order.
- `listIterator()` returns a `java.util.ListIterator` object to access the entries.
- `listIterator(int index)` returns a `java.util.ListIterator` object to access the entries. The `ListIterator` object starts from the specified position in the list.

Both the `Iterator` object and the `ListIteraor` object provide the means for iterating through the elements of the list and remove elements from the list. The `ListIterator` object allows you the additional capabilities of traversing the list in both directions and modifying elements in the list. Any changes made to elements using either the `ListIterator` object are reflected in the values stored in the original persistent list.

The `Iterator` object and the `ListIterator` object are implemented using Berkeley DB cursors. When the `Iterator` object or `ListIterator` object is created the database layer creates a read lock on the underlying datastore. This read lock is held until the iterator is closed by the database manager using the database manager's static `closeIterator()` method. `closeIterator()` takes the iterator to be closed as a parameter.

Example 263 shows code for iterating through a list.

Example 263: *Iterating through a Persistent List*

```
Iterator iter = floatlist.iterator();

while (iter.hasNext())
{
  Float entry = (Float)iter.next();
  System.out.println(Float.floatValue());
}

DatabaseManager.closeIterator(iter);
```

Closing a persistent list

When you are finished working with a persistent list, your application needs to invoke the persistent list's `close()` method. `close()` informs the database layer to release any resources used to maintain the connection to the physical representation of the datastore and flushes any buffered writes to the physical disk.

Example 259 shows code for closing a persistent list.

Example 264:*Closing a Persistent List*

```
floatMap.close();
```

Other operations

Artix persistent lists implement all of the methods of the `java.util.List` interface. These methods provide means for querying the list to see if it contains a specific object. They also provide a means for seeing if the list has any data stored in it and for converting the data into an array. For a full list of all the methods available see the Java 1.4.2 API documentation for `java.util.List` (http://java.sun.com/j2se/1.4.2/docs/api/java/util/List.html).

Note: Artix persistent lists throw an unsupported exception when invoking the `size()` method.

Configuring Artix to Use Persistent Datastores

Overview

Artix will automatically create all of the artifacts needed to use persistent datastores without adding any configuration to your Artix environment. However, Artix can be configured to control the location and name of the Berkeley DB artifacts used by the database layer.

Also, if you intend to deploy a service as a highly available cluster, that is all done in Artix configuration.

Database layer configuration

The database layer is configured using two configuration variables:

- `plugins:artix:db:env_name` specifies the filename for the Berkeley DB environment file. It can be any string and can have any file extension.
- `plugins:artix:db:home` specifies the directory where Berkeley DB stores all the files for the service databases. Each service should have a dedicated folder for its data stores. This is especially important for replicated services.

Example

Example 265 shows a configuration fragment for a service using persistent datastores.

Example 265: *Persistent Datastore Configuration*

```
# Artix Configuration File
...
foo_service {
    plugins:artix:db:env_name = "myDB.env";
    plugins:artix:db:home = "/etc/dbs/foo_service";
};
```

CHAPTER 16

Using Transactions in Artix

Artix combines a technology neutral API and pluggable transaction manager support to facilitate transactions over a wide range of transports.

In this chapter

This chapter discusses the following topics:

Introduction to Transactions in Artix

Overview

To maintain the loosely coupled and technology-neutral nature of a service oriented architecture, Artix provides a technology-neutral Java API for developing clients that request transacted services and for developing transactional Artix services. In addition, Artix's pluggable architecture allows you to use a number of underlying transaction management implementations to fit the topology of your service environment.

Supported transaction systems

Using Artix's pluggable framework, you can replace the underlying transaction system used by Artix. Currently, Artix provides support for the transaction systems:

- WS-AtomicTransactions—A light weight transaction management system based on the WS-AT specification. It supports the distributed two-phase commit protocol.
- OTS Lite—A CORBA based transaction system that provides one-phase commit.
- OTS Encina—A CORBA based transaction system that provides one-phase and two-phase commit.

Supported topologies

Artix supports distributed transactions using the following combinations of protocols and transaction managers:

- SOAP over any transport using WS-AtomicTransactions.
- CORBA using the WS-Coordination service.

Note: When the WS-Coordination service is used to coordinate CORBA transactions an OTS transaction context is used instead of a WS-AtomicTransactions context.

- CORBA using OTS Encina or OTS Light.

- SOAP over any transport using OTS Encina or OTS Light.

Note: When OTS is used as the transaction service in conjunction with SOAP messages the following conditions hold true:

- The transaction contexts are WS-AT compliant.
- OTS assumes the role of the coordinator as specifies in the WS-Coordination specification.
- Communication between all of the OTS coordinators use IIOP.

Client-side transaction support

Transaction demarcation functions can be used on the client side to initiate and terminate a transaction. While the transaction is active, all of the operations called from the current thread are included in the transaction and the operations' request headers will include a transaction context.

Server-side transaction support

On the server side, Artix provides an API that enables you to implement *transaction participants*. A transaction participant, also know as a transaction resource, is responsible for maintaining data integrity during a transaction. It is responsible for ensuring that any data modified as part of the transaction can be safely committed or rolled-back depending on the outcome of the transaction. The transaction participants enforce the ACID transaction properties (*Atomicity*, *Consistency*, *Integrity*, and *Durability*).

You only need to implement a transaction participant if your server maintains its own datastore. If your server interacts with a database system that has its own transaction resource manager, you can enlist that resource to maintain the ACID transaction properties.

One-phase commit

Artix supports the one-phase commit (1PC) protocol for transactions. This protocol can be used if there is only one resource participating in the transaction. The 1PC protocol essentially delegates the transaction completion to the single resource manager. Figure 13 shows a schematic overview of the 1PC protocol for a simple client-server system.

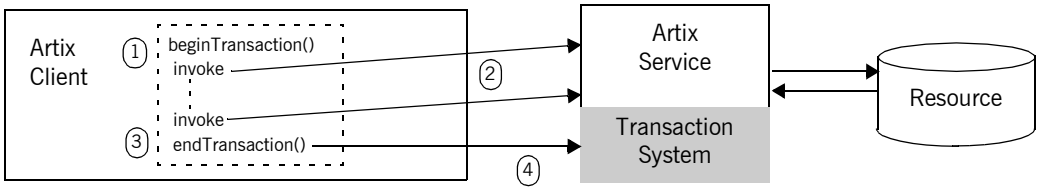

Figure 13: *One-Phase Commit Protocol*

The 1PC protocol progresses through the following stages:

1. The client calls `begin_transaction()` to initiate the transaction.
2. Within the transaction, the client calls one or more WSDL operations on the remote server.
3. The client calls `commit_transaction()` or `rollaback_transaction()` to end the transaction.
4. The transaction system sends a notification to the server instructing it to perform a 1PC commit or roll-back the transaction.

Two-phase commit

The two-phase commit (2PC) protocol enables multiple resources to participate in a transaction. In order to preserve the essential properties of a transaction involving multiple distributed resources, it is necessary to use a more elaborate algorithm. The 2PC algorithm consists of the following two phases:

- *Prepare phase*—the transaction system notifies all of the participants to prepare the transaction. The participants prepare the transaction by saving the information that would be required to redo or undo the changes made during the transaction. At the end of this phase, the participants vote whether to commit or roll back the transaction.

- *Commit (or rollback) phase*—if all of the participants vote to commit the transaction, the transaction system notifies the participants to commit the changes. On the other hand, if one or more participants vote to roll back the transaction, the transaction system notifies the participants to roll back the changes.

Figure 14 shows a schematic overview of the 2PC protocol for a client and two remote servers.

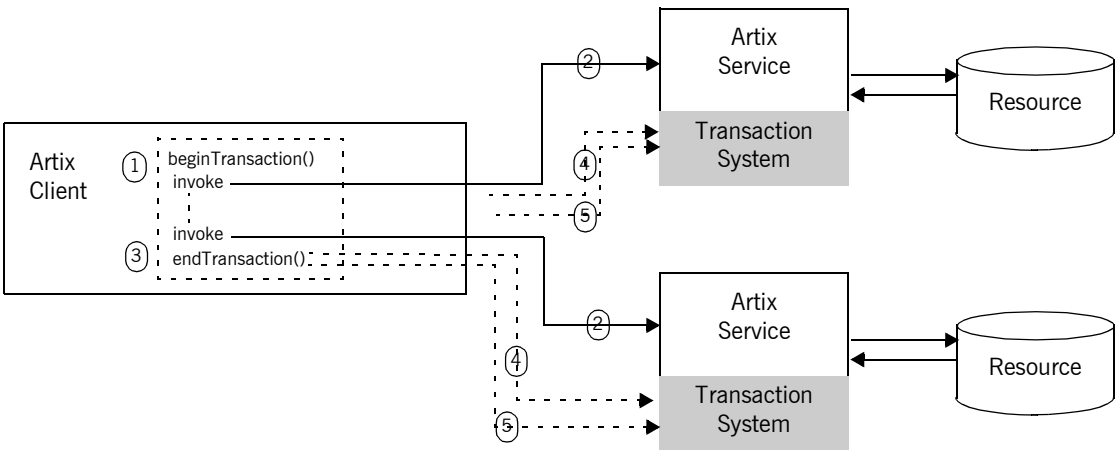

Figure 14: *Two-Phase Commit Protocol*

The 2PC protocol progresses through the following stages:

1. The client calls `begin_transaction()` to initiate the transaction.
2. Within the transaction, the client calls one or more WSDL operations.
3. The client calls `commit_transaction()` or `rollback_transaction()` to end the transaction.
4. The transaction system performs the prepare phase by polling all of the remote transaction participants.
5. The transaction system performs the commit or rollback phase by sending a notification to all of the remote transaction participants.

Transaction propagation

When an Artix server is involved in a transaction, it will automatically propagate the transaction context if it makes requests on another server. Therefore, the next service in the chain will be coordinated as part of the original transaction. When properly configured, Artix will propagate SOAP transactions over CORBA and vice versa.

Consider the scenario depicted in Figure 15.

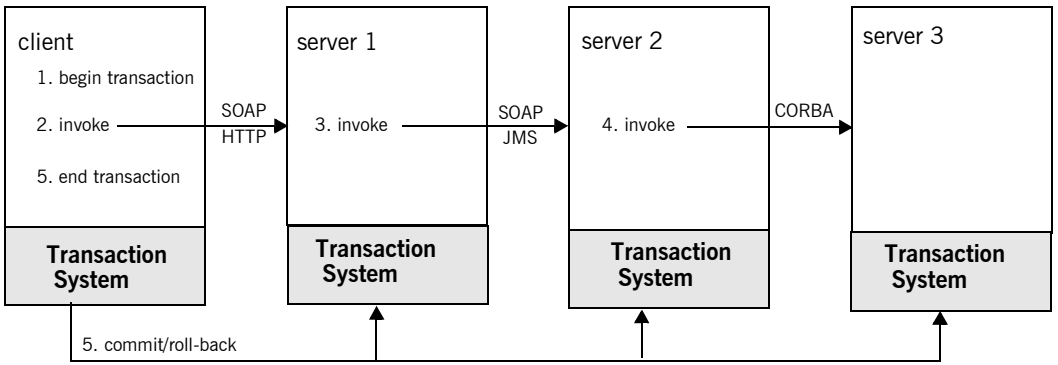

Figure 15: *Propagating Transactions Across Multiple Services*

Requests flow through the system as listed below:

1. A client initiates a transaction.
2. It invokes an operation on server 1.
3. Server 1 invokes an operation on server 2.
4. Server 2 invokes an operation on server 3.
5. Client ends the transaction.
6. Any transactional resources enlisted in servers 1, 2, & 3 are informed if they should commit the transaction or roll it back.

In this scenario, Artix automatically propagates the transaction context from server 1 to server 2. Artix propagates the context even though the transport is changed. Artix also propagates the transaction context along to server 3 even though the binding is changed.

All operation flow through the three servers is executed as part of the same transaction. During a transport or binding change, if necessary, an alternative transaction coordinator may be interposed as a subordinate to the original transaction coordinator in use when the client initiated the transaction. However, the client's coordinator is responsible for coordinating when the transaction is committed or rolled back.

WARNING: Bindings other than SOAP and CORBA do not support transaction contexts. If an intermediate server makes a request using a binding that does not support transaction contexts, any transactional processing that takes place as part of the request is not part of the transaction.

Artix also enlists MQ in transactions when the MQ transactional setting is activated. This means that any request sent over MQ as part of a transaction is executed as a local transaction. If any of the local MQ transactions fail, the MQ transaction manager will vote to roll back the transaction. For more information see "Enlisting WebSphereMQ Transactions" on page 458.

Combining middlewares inside a transaction

An Artix endpoint can be configured to coordinate transactions that include request made over a number of different transports. For example, if you were developing a system that handles direct deposits for your employees, you may be faced with a situation where the personnel records are stored in an Websphere MQ system, your company's bank uses a CORBA based system, and the employee's bank uses a Tuxedo system. To ensure the integrity of the employee's data you need to be able to invoke operations on all three systems within a transaction as shown in Figure 16. For your client to work, you would need to deploy it with both an OTS Encina transaction

system and a WS-Atomic Transactions transaction system. The OTS Encina system will handle the CORBA service and the WS-Atomic Transactions system will handle the SOAP systems.

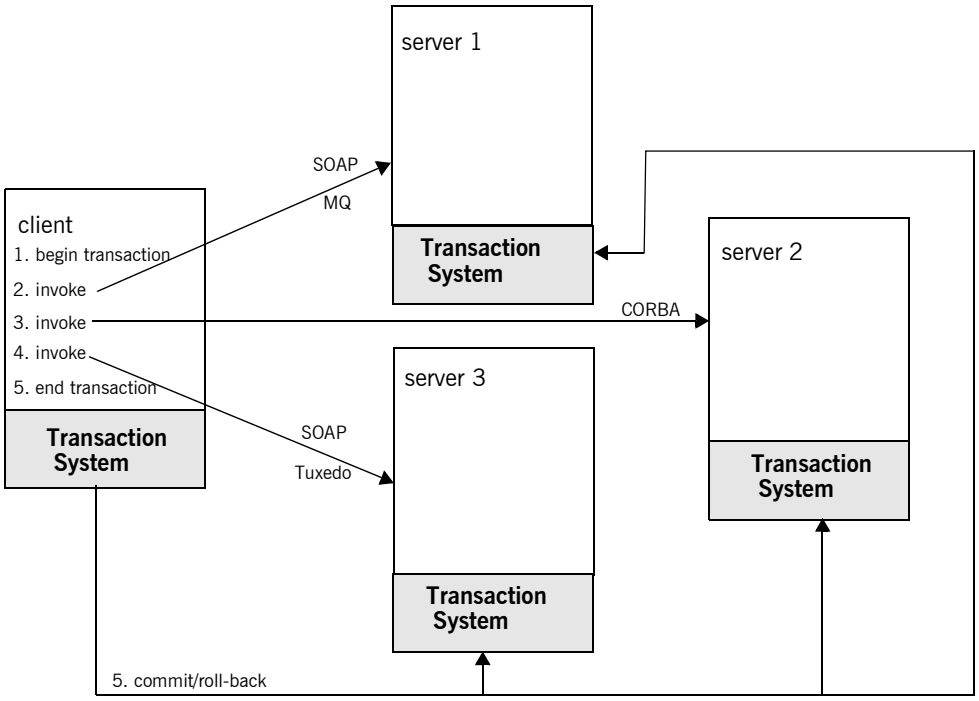

Figure 16: *Coordinating Transactions Across Multiple Middlewares*

In this scenario, the following happens:

1. The client begins the transaction and creates a transaction context to be sent out with each request.
2. The client makes a request on service 1.
   i. The client's transaction system adds a WS-AT transaction context in the SOAP header.
   ii. Service 1 reads the WS-AT transaction context.

iii. Service 1's transaction system enlists with the client's WS-Atomic Transactions coordinator.

3. The client makes a request on service 2.
   i. The client's transaction system includes the transaction context in the CORBA request.
   ii. Service 2 sees that it is part of a transaction.
   iii. Service 2's transaction system enlists with the client's OTS Encina transaction coordinator.
4. The client makes a request on service 3 and adds a WS-AT transaction context in the SOAP header.
   i. The client's transaction system adds a WS-AT transaction context in the SOAP header.
   ii. Service 3 reads the WS-AT transaction context.
   iii. Service 3's transaction system enlists with the client's WS-Atomic Transactions coordinator.
5. The client ends the transaction.

The client's transaction system informs all of the enlisted services to either commit or roll-back.

Selecting a Transaction Coordinator

Overview

Artix's plug-in architecture coupled with Artix's transaction system agnostic API allows you to choose between a number of different transaction system implementations. You can choose the transaction system that provides the best match for your services. For example, if the majority of your services are SOAP-based, you would probably select the WS-AT transaction system because it is a commonly used standard in developing Web services.

This section describes how to configure an application to use each of the transaction systems supported by Artix.

In this section

This section contains the following subsections:

Configuring OTS Lite

Overview

OTS Lite is a lightweight, CORBA based transaction manager. It is subject to the following restrictions:

- it only supports the 1PC protocol.
- it lets you register only one resource.

OTS Lite allows applications that only access a single transactional resource to use the OTS APIs without incurring a large overhead, but allows them to migrate easily to the more powerful 2PC protocol by switching to the full OTS Encina transaction system.

Like many of the features provided by Artix, OTS Lite is implemented as a plug-in that is loaded when the endpoint is started. To use OTS Lite, you need to configure your endpoints to load the generic OTS plug-in. In addition, your client endpoint needs to load the OTS Lite plug-in. Figure 17 shows a client-server deployment that uses the OTS Lite plug-in.

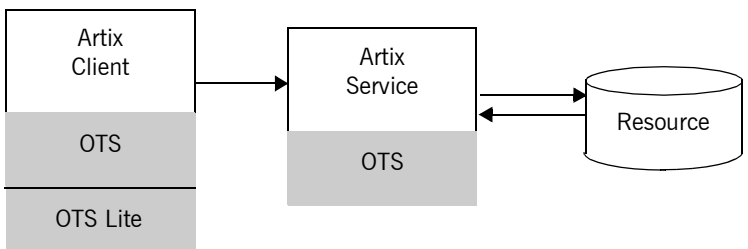

Figure 17: *Overview of a Client-Server System that Uses OTS Lite*

Limitations of using OTS-Lite

Because OTS-Lite can only enlist a single resource, it is not a viable choice when in a system where transaction contexts must cross between middleware boundaries. This is because the OTS transaction manager automatically enlists the WS-AT transaction manager when a transaction

crosses from an OTS transaction system to a WS-AT transaction system. Because the WS-AT transaction manager will most likely need to enlist a resource, OTS-Lite will not suffice.

OTS-Lite is also insufficient when transactions are propagated across multiple services, including the Artix router. Each transactional service will have its own transaction manager and as the transaction is propagated beyond the second tier, the transaction managers are each enlisted as resources by the originating transaction manager. Therefore, as soon as a transaction is propagated beyond the first service, OTS-Lite would become unable to enlist the additional transaction managers.

Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the CORBA OTS transaction system, you must initialize this configuration variable with the value, `ots_tx_provider`.

Loading the OTS plug-in

In order to use the CORBA OTS transaction system, the OTS plug-in must be loaded both by the client and by the server. To load the OTS plug-in, include the `ots` plug-in name in the `orb_plugins` list. For example:

```
ots_lite_client_or_server {
    plugins:bus:default_tx_provider:plugin = "ots_tx_provider";
    orb_plugins = [ ..., "ots"];
};
```

Loading the OTS Lite plug-in

The OTS Lite plug-in, which is capable of managing 1PC transactions, is required for any endpoints that make service requests. You can load the OTS Lite plug-in in one of the following ways:

- *Dynamic loading*—configure Artix to load the `ots_lite` plug-in dynamically, if it is required. For this approach, you need to configure the `initial_references:TransactionFactory:plugin` variable as follows:

```
ots_lite_client_or_server {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots"];
  initial_references:TransactionFactory:plugin = "ots_lite";
  ...
};
```

  This style of configuration has the advantage that the OTS Lite plug-in is loaded only if it is actually needed.

- *Explicit loading*—load the `ots_lite` plug-in by adding it to the list of `orb_plugins`, as follows:

```
ots_lite_client {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots", "ots_lite"];
  ...
};
```

Sample configuration

The following example shows a sample configuration for using the OTS Lite transaction manager:

```
# Basic configuration for transaction plug-ins (shared library
#   names and so on) included in the global configuration scope.
#   ... (not shown)

ots_lite_client_or_server {
    plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop", "ots"];
    initial_references:TransactionFactory:plugin = "ots_lite";
};
```

Configuring OTS Encina

Overview

The Encina OTS Transaction Manager provides full recoverable 2PC transaction coordination implemented on top of the industry proven Encina Toolkit from IBM/Transarc. Encina supports both 1PC and 2PC protocols and allows you to register multiple resources. Like the OTS Lite transaction manager, it is CORBA based and exposes the OTS APIs.

The OTS Encina transaction system is implemented as a plug-in that is loaded when the endpoint is started. To use OTS Encina, you need to configure your endpoints to load the generic OTS plug-in. In addition, your client endpoint needs to load the OTS Encina plug-in. Figure 18 shows a client/server deployment that uses the OTS Encina plug-in.

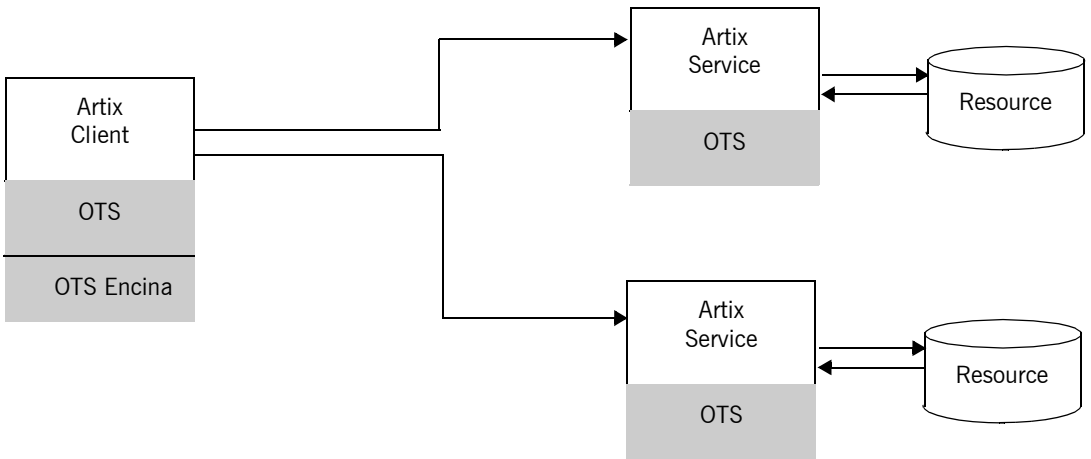

Figure 18: *Overview of a Client-Server System that Uses OTS Encina*

Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the CORBA OTS transaction system, you must initialize this configuration variable with the value, `ots_tx_provider`.

Loading the OTS plug-in

For applications that use the CORBA OTS transaction system, the OTS plug-in must be loaded both by the client and by the server. To load the OTS plug-in, include the `ots` plug-in name in the `orb_plugins` list. For example:

```
# Artix Configuration File
ots_encina_client_or_server {
    plugins:bus:default_tx_provider:plugin = "ots_tx_provider";
    orb_plugins = [ ..., "ots"];
};
```

Loading the OTS Encina plug-in

The OTS Encina plug-in, which is capable of managing 1PC and 2PC transactions, is required for any endpoint that makes service requests. You can load the OTS Encina plug-in in one of the following ways:

- *Dynamic loading*—configure Artix to load the `ots_encina` plug-in dynamically, if it is required. For this approach, you need to configure the `initial_references:TransactionFactory:plugin` variable as follows:

  ```
  # Artix Configuration File
  ots_encina_client_or_server {
    plugins:bus:default_tx_provider:plugin="ots_tx_provider";
    orb_plugins = [ ..., "ots"];
    initial_references:TransactionFactory:plugin="ots_encina";
    ...
  };
  ```

  This style of configuration has the advantage that the OTS Encina plug-in is loaded only if it is actually needed.

- *Explicit loading*—load the `ots_encina` plug-in by adding it to the list of `orb_plugins`, as follows:

  ```
  # Artix Configuration File
  ots_lite_client {
    plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
    orb_plugins = [ ..., "ots", "ots_encina"];
    ...
  };
  ```

Sample configuration

Example 266 shows a complete configuration for using the OTS Encina transaction manager:

Example 266: *Sample Configuration for OTS Encina Plug-In*

```
# Artix Configuration File
ots_lite_client_or_server {
1       plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
        orb_plugins = [ ..., "ots"];

2       initial_references:TransactionFactory:plugin = "ots_encina";

3       plugins:ots_encina:direct_persistence = "true";

4       plugins:ots_encina:initial_disk = "../../log/encina.log";
5       plugins:ots_encina:initial_disk_size = "1";
6       plugins:ots_encina:restart_file =
       "../../log/encina_restart";
7       plugins:ots_encina:backup_restart_file =
       "../../log/encina_restart.bak";

        # Boilerplate configuration settings for OTS Encina:
        # (you should never need to change these)
8       plugins:ots_encina:shlib_name = "it_ots_encina";
        plugins:ots_encina_adm:shlib_name = "it_ots_encina_adm";
        plugins:ots_encina_adm:grammar_db =
       "ots_encina_adm_grammar.txt";
        plugins:ots_encina_adm:help_db = "ots_encina_adm_help.txt";
};
```

The preceding configuration can be described as follows:

1. These two lines configure Artix to use the CORBA OTS transaction system and load the OTS plug-in.
2. This line configures Artix to load the `ots_encina` plug-in dynamically, if it is needed by the application (typically needed on the client side).
3. Configuring Encina to use direct persistence means that the Encina transaction manager service listens on a fixed IP port.
4. The `plugins:ots_encina:initial_disk` variable specifies the path for the initial file used by the Encina OTS for its transaction logs.

   If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).

5. The `plugins:ots_encina:initial_disk_size` variable specifies the size of the initial file used by the Encina OTS for its transaction logs. Defaults to 2.
6. The `plugins:ots_encina:restart_file` variable specifies the path for the restart file, which Encina OTS uses to locate its transaction logs.

   If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).
7. The `plugins:ots_encina:backup_restart_file` variable specifies the path for the backup restart file, which Encina OTS uses to locate its transaction logs.

   If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).
8. The settings in the next few lines specify the basic configuration of the OTS Encina plug-in. It should not be necessary ever to change the values of these configuration settings.

Configuring WS-Atomic Transactions

Overview

The WS-Atomic Transactions (WS-AT) transaction system is IONA's implementation of the WS-Atomic Transactions specification (ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf). It uses SOAP headers to transmit transaction contexts between the participants in a transaction. The WS-AT transaction system supports the 2PC protocol and allows you to register multiple resources.

The WS-AT transaction system is implemented as a group of plug-ins:

- WS-AT plug-in - This plug-in is responsible for creating a WS-AT transaction context, registering it with the transaction coordinator, and adding the WS-AT transaction context to the SOAP headers of transactional requests. It is also responsible for reading the transaction context from incoming requests and interacting with the transaction coordinator.
- WS-Coordinator plug-in - This plug-in coordinates the resources enlisted in a transaction. It coordinates the voting of the resources and the implementation of the votes outcome.

All clients and services that use the WS-AT transaction system to coordinate transactional flows must load the WS-AT plug-in. Without it, they will not be able to handle the WS-AT transaction context that is required. Only clients that start transactions need to be able to access the features of the WS-Coordination plug-in. The clients can either directly load the

WS-Coordination plug-in as shown in Figure 19. An alternative approach is to load the WS-Coordination plug-in into an instance of the Artix container service and run as a standalone service.

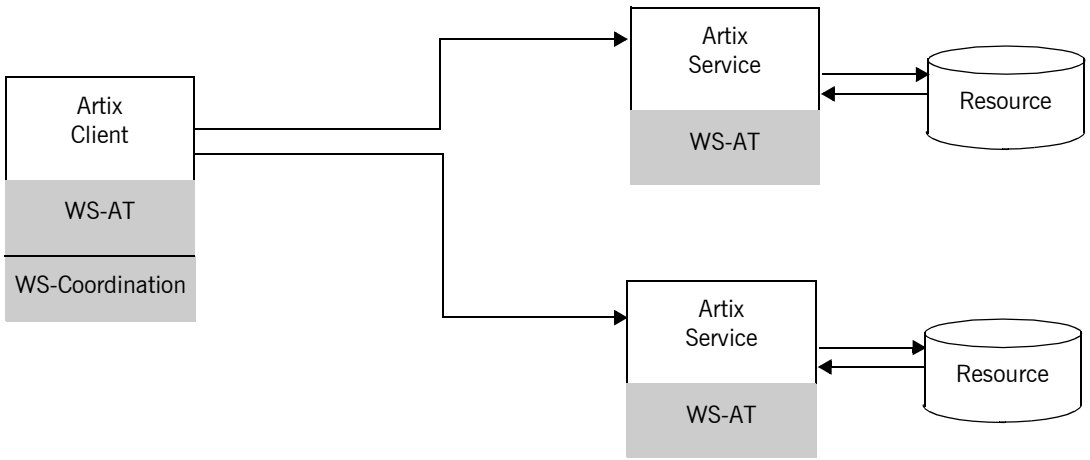

Figure 19: *Overview of a Client-Server System that Uses WS-AT*

Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the WS-AT transaction system, you must initialize this configuration variable with the value, `wsat_tx_provider`.

Plug-ins for WS-AT

The division of the WS-AT transaction system into separate plug-ins reflects the fact that the WS-AT specification has two distinct parts: WS-AtomicTransactions and WS-Coordination.

The following plug-ins are required to support the WS-AT transaction system:

- `wsat_protocol` plug-in—implements the WS-Atomic Transactions specification. It is required by all endpoints that use WS-AT transactions. This plug-in enables an Artix executable to receive and transmit WS-AT transaction contexts.
- `ws_coordination_service` plug-in—implements the WS-Coordination specification(ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf). An instance of this plug-in is required to be running and it must be accessable by endpoints that make service requests. This plug-in coordinates the two-phase commit protocol.
- `coordinator_stub_wsdl` plug-in—allows a transactional service to interact with the WS-Coordination plug-in.

Sample configuration

Example 267shows a complete configuration for using the WS-AT transaction manager:

Example 267:*Sample Configuration for WS-AtomicTransactions*

```
# Artix Configuration File
ws_atomic_transactions {
  client
  {
1   orb_plugins = ["local_log_stream",
                   "ws_coordination_service"];
2   plugins:bus:default_tx_provider:plugin ="wsat_tx_provider";
    };

  server
  {
3   orb_plugins = ["local_log_stream", "wsat_protocol",
                   "coordinator_stub_wsdl"];
  }
};
```

The preceding configuration can be described as follows:

1. The `ws_coordination_service` plug-in is needed only on the client side. Artix does *not* support auto-loading of this plug-in.

   The `ws_coordination_service` plug-in implicitly loads the `wsat_protocol` plug-in as well. Hence, it is unnecessary to include `wsat_protocol` plug-in in the `orb_plugins` list on the client side.
2. This line specifies that WS-AT is the default transaction provider. This implies that whenever a client initiates a transaction (for example, by calling `begin_transaction()`), Artix creates a new WS-AT transaction by default.
3. The server needs to load the `wsat_protocol` plug-in, in order to process incoming atomic transactions coordination contexts and to propagate transaction contexts. The `coordinator_stub_wsdl` plug-in enables the server to talk to the WS-Coordination service on the client side.

Transaction API

Overview

Figure 20 shows an overview of the main classes that make up the Artix transaction API. The Artix transaction API is designed to function as a generic wrapper for a wide variety of specific transaction systems. As long as you use the Artix APIs, you will be able to switch between any of the transaction systems supported by Artix.

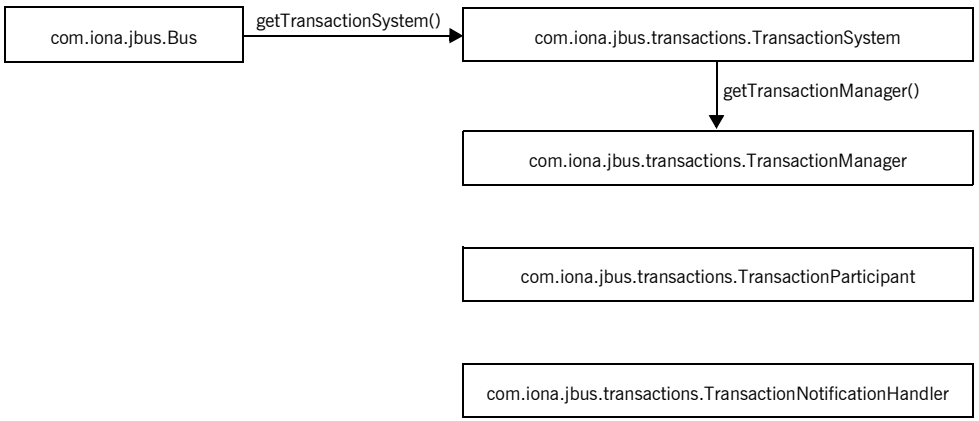

Figure 20: *Overview of the Artix Transaction API*

Accessing the transaction system

To access the Artix transaction system, call the `getTransacionSystem()` method on the bus. The returned `com.iona.jbus.transaction.TransactionSystem` object provides the starting point for accessing all aspects of Artix transactions.

The signature of `Bus.getTransactionSystem()` is shown in Example 268.

Example 268: *Signature for getTransactionSystem()*

```
TransactionSystem getTransactionSystem() throws BusException;
```

TransactionSystem class

The `TransactionSystem` class provides the basic methods needed for transaction demarcation (`beginTransaction()`, `commitTransaction()` and `rollbackTransaction()`). For more details see "Beginning and Ending Transactions" on page 440.

In addition to providing access the transaction demarcation method the `TransactionSystem` object provides two other methods:

- `getTransactionManager()` returns a `com.iona.jbus.transaction.TransactionManager` object that provides access to some of the more advanced transaction features.
- `withinTransaction()` returns true if it is called within an active transaction.

TransactionManager class

The `TransactionManager` class provides advanced transaction functionality. The most important method it provides is `enlist()`, which enables you to implement a transactional resource by enlisting a transaction participant object. It also provides methods for attaching and detaching threads from a transaction. See "Threading" on page 451.

TransactionParticipant interface

The `com.iona.jbus.transaction.TransactionParticipant` interface is used to create transactional resources. An implementation of `TransactionParticipant` acts as the resource manager for the datastore involved in the transaction. It receives callbacks from the transaction manager that are used to coordinate the commit or rollback steps with other transaction participants. For more details, see "Managing Transactional Resources" on page 445.

TransactionNotificationHandler interface

The `com.iona.jbus.transacation.TransactionNotificationHandler` interface is used to create objects that receive notification callbacks from the transaction manager whenever a transaction is either committed or rolled back.

Beginning and Ending Transactions

Overview

The functions for beginning, committing, and rolling back a transaction are collectively referred to as *transaction demarcation* functions. Within a given thread, any Artix operations invoked after the transaction *begin* and before the transaction *commit*, or *rollback*, are associated with the transaction. The transaction demarcation functions are typically the only functions that you need on the client side.

Procedure

When an endpoint needs to enclose a group of operations inside a transaction, the following steps are taken:

1. Get an instance of the `TransactionSystem` from the bus.
2. Begin the transaction by calling `TransactionSystem.beginTransation()`.
3. Invoke the operations that make up the transaction.
4. If all operations are completed successfully, commit the transaction by calling `TransactionSystem.commitTransaction()`.
5. If there is a problem during the transaction, roll-back the transaction by calling `TransactionSystem.rollbackTransaction()`.

Getting the transaction system

In order to work with transactions you must first get access to the transaction system. This is done by calling the bus' `getTransactionSytem()` method. `getTransacitonSystem()` returns a `TransactionSystem` object that provides the methods needed to being and end transactions. Example 269 shows code for getting access to the transaction system.

Example 269: *Getting Access to the Transaction System*

```
import com.iona.jbus.Bus;
import com.iona.jbus.transaction.*;

Bus bus = Bus.init(args);
TransactionSystem txSystem = bus.getTransactionSystem();
```

Beginning a transaction

You start a transaction using the `TransactionSystem.beginTransaction()` method. `beginTransaction()` takes no arguments and doesn't have a return value. `beginTransaction()` can throw the following exceptions:

- `TransactionAlreadyActiveException` is thrown if `beginTransaction()` is called inside an active transaction.
- `TransactionSystemUnavailableException` is thrown if the transaction system cannot be loaded. This usually points to a configuration problem.

Once `beginTransaction()` has been called all requests will include a transaction context and they will be managed as one atomic transaction until the transaction is concluded.

Example 270 shows code for starting a transaction.

Example 270: *Starting a Transaction*

```
try
{
  txSystem.beginTransaction();
}
catch (TransactionSystemUnavialableException)
{
 // handle the exception
}
```

Ending a transaction

There are two ways to end a transaction. The most common is to commit it using `TransactionSystem.commitTransation()`. When a transaction is committed, the transaction manager instructs all of the enlisted resources to make the changes that occurred in processing the transaction permanent. Because instructing the transaction manager to commit a transaction does not ensure that the transaction will actually be committed, `commitTransaciton()` can returns a boolean value specifying if the commit was successful. `commitTransaction()` takes a boolean argument that specifies if heuristic decisions should be reported during the commit protocol.

Note: Heuristic decisions are not supported by all transaction systems.

The other way to end a transaction is to roll-back the transaction by calling `TransactionSystem.rollbackTransaction()`. When a transaction is rolled back, the transaction manager instructs all the enlisted resources to erase any changes made during the processing of the transaction. All of the data is returned to its original state. Transactions are typically rolled back when an error occurs while the transaction is being processed.

Example 271 shows code for committing a transaction.

Example 271: *Committing a Transaction*

```
try
{
  txSystem.commitTransaction();
}
catch (TransactionSystemUnavialableException)
{
 // handle the exception
}
```

Other transaction functions

In addition to the demarcation functions, the `TransactionSystem` class also provides the following functions:

- `withinTransaction()`—returns `true` if the current thread is associated with a transaction; otherwise, `false`.
- `getTransactionManager()`—returns an instance of a `TransactionManager` object.

Example

Example 272 shows an Artix client that invokes a series of operations as an atomic transaction. The client invokes on single service called `Data`. `Data` provides a `read` and a `write` function.

Example 272: *Transactional Client Example*

```
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.jbus.transaction.*;

public class Transaction Client
{
  public static void main(String args[]) throws Exception
  {
1   Bus bus = Bus.init(args);
2   String serviceName = "DataService";
    String wsdlName = "soap_tx_demo.wsdl";
    QName serviceQName = new QName("http://transaction_demo",
                                   serviceName);
    QName portQName = new QName("", "DataSOAPPort");
    Data client = null;
    URL wsdlLocation = new URL(wsdlName);
    ServiceFactory factory = ServiceFactory.newInstance();
    Service service = factory.createService(wsdlLocation,
                                            serviceQName);
    client = (Data)service.getPort(portQName,Data.class);

3   TransactionSystem txSystem = bus.getTransactionSystem();

4   txSystem.beginTransaction();
```

Example 272:*Transactional Client Example*

```
    try
    {
      int value = client.read();
      System.out.println("value: " + value);
      System.out.println("Incrementing the value" );
      client.write(value + 1);
      System.out.println("New values are" );
      int value2 = client.read();
      System.out.println("value: " + value2);
    }
    catch (Trowable T)
    {
      System.out.println("rolling back transaction...");
      txSystem.rollbackTransation();
      System.exit(1);
    }
    System.out.println("committing transaction...");
    boolean result = txSystem.commitTransaction(true);
    if (result)
    {
      System.out.println("Transaction committed!");
    }
    else
    {
      System.out.println("Transaction *not* Committed!!");
    }
  }
}
```

The code in Example 272 does the following:

1. Initializes the bus.
2. Creates a proxy for the `Data` service.
3. Gets the transaction system.
4. Begins a transaction.
5. Invokes operations on the service.
6. Rolls back the transaction if an exception is thrown while invoking operations on the service.
7. Commits the transaction if all of the operations succeeded.

Managing Transactional Resources

Overview

An Java based Artix service that wants to use a transactional resource such as a database, it needs to implement a *transaction participant*. A transaction participant is an object that interfaces between the transaction manager and a persistent resource. The role of the transaction participant is to receive instructions from the transaction manager, which tell the participant whether to make pending changes permanent or whether to abort the current transaction and return the resource to its previous consistent state.

Participants in a 2-phase commit

Figure 21 shows an example of a two-phase commit involving two transaction participant instances. Any operations meant to be transactional should start by creating a transaction participant object and enlisting it with the transaction manager.

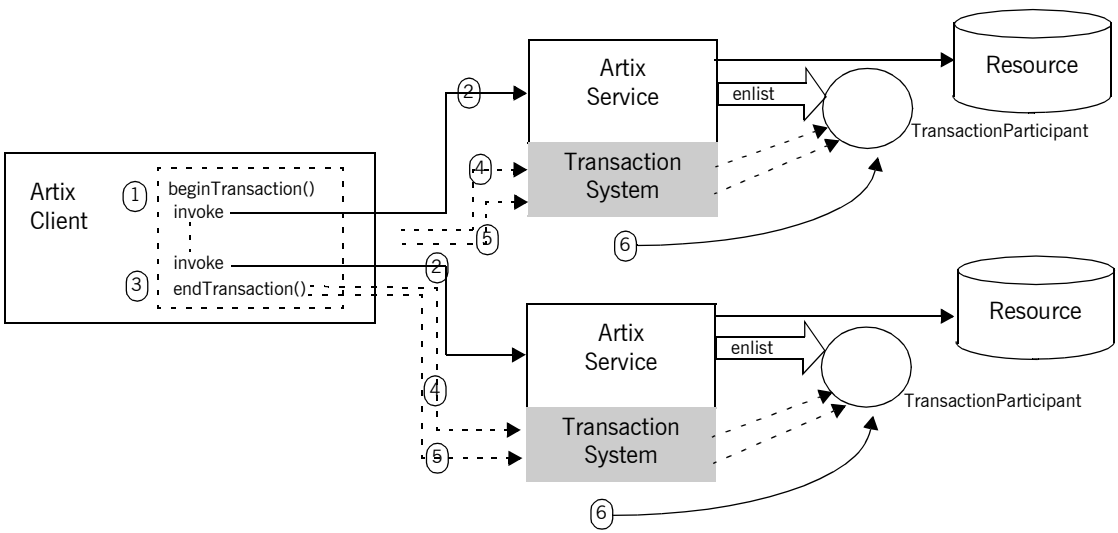

Figure 21: *Transaction Participants in a 2-Phase Commit Protocol*

As shown in Figure 21, the transaction participants participate in a two-phase commit as follows:

1. The client calls `beginTransaction()` to initiate a distributed transaction.
2. Within the transaction, the client invokes operations on Server A and on Server B. In order to participate in the distributed transaction, the servant code creates a new transaction participant and enlists it with the transaction manager.
3. The client calls `commitTransaction()` to make permanent any changes caused during the transaction.
4. The transaction system performs the prepare phase by calling `prepare()` on all of the transaction participants.
5. The transaction system performs the commit or rollback phase by calling `commit()` or `rollback()` on all of the transaction participants.
6. When the transaction is finished, the transaction manager automatically deletes the associated transaction participant instances.

Implementing a transaction participant

To create a transaction participant, define a class that implements the `com.iona.jbus.transaciton.TransactionParticipant` interface.

Example 273 shows the `TransactionParticipant` interface.

Example 273: *The TransactionParticipant Interface*

```
package com.iona.jbus.transaction;

import com.iona.jbus.BusException;

public interface TransactionParticipant
{
  void commitOnePhase() throws BusException;

  VoteOutcome prepare();

  void commit();

  void rollback();

  void setTransactionManager(TransactionManager txManager);

  String preferredTransactionManager();
}
```

1PC callback function

The following function is called during a one-phase commit:

- `commitOnePhase()`—makes permanent any changes associated with the current transaction.

2PC callback functions

The following functions are called during a two-phase commit:

- `prepare()`—called during *phase one* of a two-phase commit. Before returning, this function should write a recovery log to persistent storage. The recovery log should contain whatever data would be necessary to restore the system to a consistent state, in the event that the server crashes before the transaction is finished.

  Note: In some transaction systems, such as OTS Encina, the transaction manager will not call `prepare()` if it knows that transaction will be rolled back.

  The `prepare()` function also votes on whether to commit or roll back the transaction overall, by returning one of the following vote outcomes:

- `VoteOutcome.VOTE_CONMMIT`—vote to commit the transaction.
- `VoteOutcome.VOTE_ROLLBACK`—vote to roll back the transaction. For example, you would return `VOTE_ROLLBACK`, if an error occurred while attempting to write the recovery log.
- `VoteOutcome.VOTE_READONLY`—explicitly request *not* to be included in the commit phase of the 2PC protocol.

- `commit()`—called during *phase two* of a two-phase commit, if the transaction outcome was successful overall. The implementation of this function should make permanent any changes associated with the current transaction.
- `rollback()`—called during *phase two* of a two-phase commit, if the transaction must be aborted. The implementation of this function should undo any changes associated with the current transaction, returning the system to the state it was in before.

Getting the transaction manager

After the transaction participant is enlisted by a transaction manager instance, the transaction system calls back to pass a transaction manager to the participant. The following functions are relevant to this callback behavior:

- `preferredTransactionManager()`—called just after the participant is enlisted. The return value is a string that tells the transaction system what type of transaction manager the participant requires. The following return strings are supported:
  - `DEFAULT_TRANSACTION_TYPE`—no preference; use the current default.
  - `OTS_TRANSACTION_TYPE`—prefer the `OTSTransactionManager` interface.
  - `WSAT_TRANSACTION_TYPE`—prefer the `WSATTransactionManager` interface.
- `setTransactionManager()`—called after the `preferredTransactionManager()` call. The transaction system calls `setTransactionManager()` to pass a transaction manager of the preferred type to the participant. If the type of transaction manager requested by the participant differs from the one currently in use, Artix uses *interposition* to simulate the preferred transaction manager type.

Enlisting a transaction participant

Example 274 shows an example of how to enlist a participant instance in a transaction. You must enlist a participant at the start of any transactional WSDL operation. Example 274 shows a sample implementation of an operation, `write()`, which is called in the context of a transaction.

Example 274: *Example of Enlisting a Transactional Participant*

```
public void write(int value) throws Exception
{
  Bus bus = DispatchLocals.getCurrentBus();

  TransactionSystem txSystem = bus.getTransactionSystem();

  if (txSystem.withinTransaction())
  {
    TxParticipant participant = new TxParticipant(this);

    TransactionManager txManager =
   txSystem.getTransactionManager(TransactionSystem.DEFAULT_TRAN
   SACTION_TYPE);

    txManager.enlist(participant, true);

    m_value = value;
  }
  else
  {
    System.out.println("No transaction");
    throw new BusException("Invocation not in transaction");
  }
}
```

The preceding code example can be explained as follows:

1. `DispatchLocals.getCurrentBus()` is a standard function that returns a reference to the current thread's bus instance.
2. `write()` *requires* a transaction. If it is not called in the context of a transaction, it raises an exception back to the client.
3. The `TXParticipant` class is an implementation of the `TransactionParticipant` interface.
4. The participant is enlisted in the transaction, ensuring that the participant receives callbacks either to commit or rollback any changes.

The second parameter is a boolean flag that specifies the kind of participant:

- `true` indicates a *durable participant*, which participates in all phases of the transaction.
- `false` indicates a *volatile participant*, which is only guaranteed to participate in the prepare phase of the 2PC protocol. There is no guarantee that a volatile participant will participate in the commit phase.

Threading

Overview

Artix supports a threading API that enables you to change the thread affinity of a given transaction. Using the `attachThread()` and `detachThread()` methods, you can flexibly re-assign threads to a transaction if the underlying transaction system supports it.

Default client threading model

Figure 22 shows the default threading model for transaction. When you call `beginTransaction()`, Artix creates a new transaction and attaches it to the current thread. So long as the transaction remains attached, any WSDL operations called from the current thread become part of the transaction. When you end the transaction using either `commitTransaction()` or `rollbackTransaction()`, the transaction is deleted.

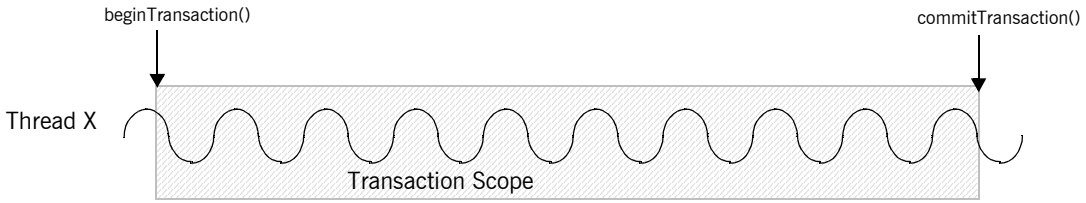

Figure 22: *Default Client Threading Model*

Transaction identifiers

A *transaction identifier* is an opaque identifier of type `com.iona.jbus.transaction.TransactionIdentifier` that uniquely identifies a transaction.

Controlling thread affinity

On the client side, thread affinity is controlled by the `TransactionManager` methods shown in Example 275.

Example 275: *Functions for Controlling Thread Affinity*

```
public class TransactionManager
{
  public TransactionIdentifier detachThread();

  public boolean attachThread(TransactionIdentifier
   transactionIdentifier)
  throws InvalidTransactionIdentifierException

  public TransactionIdentifier getTransactionIdentifier()
   ...
}
```

These functions can be explained as follows:

- `detachThread()` detachs the transaction from the current thread. After the call to `detachThread()`, WSDL operations called from the current thread do not participate in the transaction. The returned transaction identifier can be used to re-attach the transaction to the current thread at a later stage.
- `attachThread()` attachs the specifies transaction to the current thread.
- `getTransactionIdentifier()` returns the identifier of the transaction that is attached to the current thread. If no transaction is attached, it returns `NULL`.

Detaching and re-attaching a transaction to a thread

Figure 23 shows how to use the `detachThread()` and `attachThread()` functions to suspend temporarily the association between a transaction and a thread. This can be useful if, in the midst of a transaction, you need to perform some non-transactional tasks.

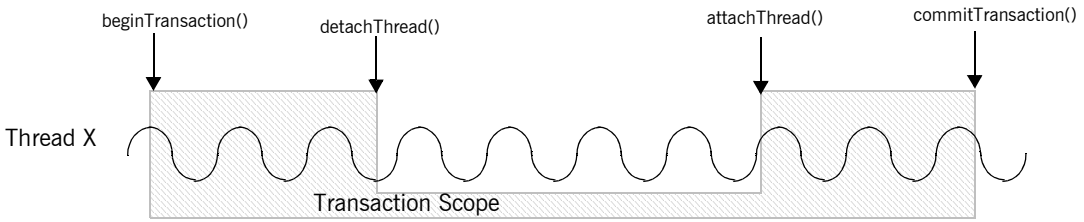

Figure 23: *Detaching and Re-Attaching a Transaction to a Thread*

Attaching a transaction to multiple threads

Figure 24 shows how to use the `getTransactionIdentifier()` and `attachThread()` functions to associate a transaction with multiple threads. The `getTransactionIdentifier()` function is called from within the thread that initiated the transaction. The transaction ID can then be passed to the other threads, Y and Z, enabling them to attach the transaction.

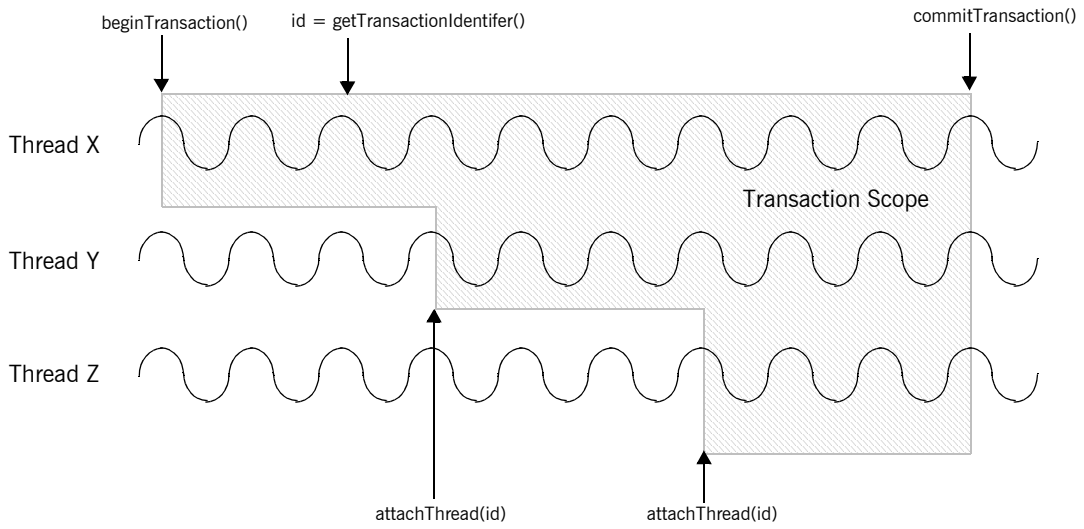

Figure 24: *Attaching a Transaction to Multiple Threads*

Note: Some transaction systems do not allow you to associate multiple threads with a transaction. In this case, an `attachThread()` call returns `false` if you attempt to attach a second thread to the transaction.

Transferring a transaction from one thread to another

Figure 25 shows how to use the `detachThread()` and `attachThread()` functions to transfer a transaction from thread X to thread Y. The transaction ID returned from the `detachThread()` call must be passed to thread Y, enabling it to attach the transaction.

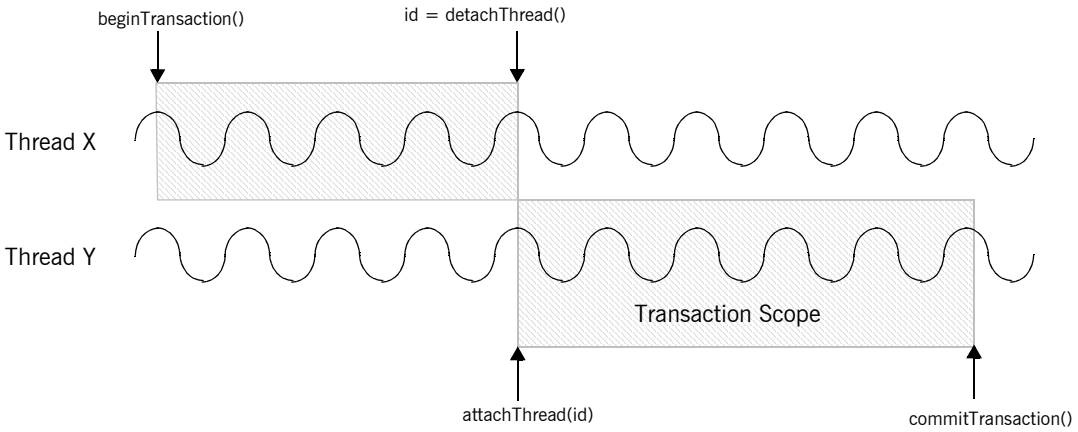

Figure 25: *Transferring a Transaction from One Thread to Another*

Note: Some transaction systems do not allow you to transfer a transaction from one thread to another. In this case, `attachThread()` returns `false` unless you are re-attaching the original thread to the transaction.

Notification Handlers

Overview

A *notification handler* is an object that records the outcome of a transaction. For example, you might use a notification handler to log transaction outcomes or to synchronize other events with a transaction.

Implementing a notification handler

To implement a notification handler, implement the `com.iona.jbus.transaction.TransactionNotificationHandler` interface.

Example 276 shows the `TransactionNotificationHandler` interface. These operations will only be called if an appropriate notification mechanism is available in the underlying transaction system.

Example 276: *The TransactionNotificationHandler Interface*

```
package com.iona.jbus.transaction;

public interface TransactionNotificationHandler
{
  void commitInitiated(TransactionIdentifier transactionId);

  void committed();

  void aborted();
}
```

Notification callback functions

The following notification handler functions receive callbacks from the transaction manager:

- `commitInitiated()`—informs the handler that a commit has been initiated. This function is called before any participants are prepared.

  Note: WS-AT does not support this notification point.

- `committed()`—informs the handler that the transaction completed successfully.
- `aborted()`—informs the handler that the transaction did not complete successfully and was aborted.

Enlisting a notification handler

To use a notification handler, you must enlist it with a `TransactionManager` object while there is a current transaction. You can enlist a notification handler at any time prior to the termination of the transaction.

Example 277 shows how to enlist a sample notification handler, `NotificationHandlerImpl`.

Example 277: *Example of Enlisting a Notification Handler*

```
Bus bus = DispatchLocals.getCurrentBus();
TransactionSystem txSystem = bus.getTransactionSystem();

if (txSystem.withinTransaction())
{
  NotificationHandlerImpl notHandler = new
   NotificationHandlerImpl;

  TransactionManager txManager =
   txSystem.getTransactionManager(TransactionSystem.DEFAULT_TRAN
   SACTION_TYPE);

  txManager.enlistForNotification(notHandler);
}
```

Enlisting WebSphereMQ Transactions

Overview

This section describes how WebSphere MQ transactions can be enlisted as part of an Artix transaction. WebSphere MQ transactions differ in several important respects from Artix transactions:

- MQ transactions are managed by a transaction manager that is internal to WebSphere MQ.
- MQ transactions are enabled by setting the relevant attributes of a WSDL port in the WSDL contract.

When a request containing a transaction context is passed over a transactional MQ queue, the MQ transaction manager is automatically enlisted in the global transaction. This means that MQ fails to deliver a message that is part of the transaction, the MQ transaction manager will vote for the entire transaction to be rolled back.

Client configuration

To enable transactional semantics for a client using the MQ transport, you should define a WSDL port as shown in Example 278.

Example 278: *WSDL Port Configuration for Oneway Client Over MQ*

```
<wsdl:service name="MQService">
    <wsdl:port binding="tns:BindingName" name="PortName">
        <mq:client ...
                   Transactional="internal"
                   Delivery="persistent"
                   UsageStyle="peer"/>
        ...
    </wsdl:port>
</wsdl:service>
```

To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Server configuration

To enable transactional semantics for a server using the MQ transport, you should define a WSDL port as shown in Example 279.

Example 279: *WSDL Port Configuration for Oneway Server Over MQ*

```
<wsdl:service name="MQService">
    <wsdl:port binding="tns:BindingName" name="PortName">
        ...
        <mq:server ...
                   Transactional="internal"
                   Delivery="persistent"
                   UsageStyle="peer"/>
    </wsdl:port>
</wsdl:service>
```

To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

CHAPTER 17

Using the Call Interface for Dynamic Invocations

The JAX-RPC Call interface allows you to make invocations on remote services for which you only have a WSDL description.

In this chapter

This chapter discusses the following topics:

DII and the Call Interface

What is DII?

DII stands for *Dynamic Invocation Interface*. DII provides a mechanism by which you can invoke on remote services without having the stubs statically linked into your application code. Using DII, you query a service for a description of its interface, use that description to dynamically build the proper invocation interface, and then use the dynamic interface to invoke on the service. This is useful if your application cannot always be sure of the exact structure of the request message or must dynamically request services from a repository of some sort.

The Call interface

The JAX-RPC specification defines the `Call` interface to support DII. Using the `Call` interface, Artix developers can invoke on remote services without needing to have access to the service's generated interface. To invoke on a remote service using the `Call` interface, you need to get a copy of the remote service's WSDL contract, a description of the message expected by the service, and any message the service may return. With this information you build, at runtime, the interface needed to invoke on the remote service and receive a response.

Artix DII support

Artix supports the majority of the functions specified in sections 8.2.4-8.2.8 of the JAX-RPC specification. The limitations are listed below.

- Artix does not support the `javax.xml.rpc.session.maintain` standard property.
- The methods listed in Table 28 are not supported by the Artix implementation of the `Service` interface.

Table 28: *Unsupported Service Methods*

Method Signature
`TypeMappingRegistry getTypeMappingRegistry();`
`HandlerRegistry getHandlerRegistry();`
`Remote getPort(Class intfc) throws ServiceException;`

Table 28: *Unsupported Service Methods*

Method Signature
`Iterator getPorts() throws ServiceException;`

- The methods listed in Table 29 are not supported by the Artix implementation of the `ServiceFactory` interface.

Table 29: *Unsupported ServiceFactory Methods*

Method Signature
`Service createService(Qname qname) throws ServiceException;`
`Service loadService(Class class1) throws ServiceException;`
`Service loadService(URL url, Class class1, Properties props) throws ServiceException;`
`Service loadService(URL url, QName qname, Properties props) throws ServiceException;`

Building Invocations using the Call Interface

Overview

Using a dynamic proxy to invoke on a remote service requires you to discover the name of the remote service's operation that you wish to invoke. It also requires you to carefully construct the parameter list for the operation. There are several ways to get this information. They range from giving the client application some foreknowledge of the possible operations it will invoke to parsing the services WSDL to recreate the operation.

Applications that use the `Call` interface to dynamically invoke on remote services also need to have knowledge of the types used by the services from which they request services. The application making the dynamic invocation must register the type factories for any complex types used by the remote services on which it will invoke. For more information on type factories see "Working with Artix Type Factories" on page 201.

Procedure

To make a dynamic service invocation using the `Call` interface do the following:

1. Register the type factories for the complex types the application may use in building a dynamic invocation. See "Registering Type Factories" on page 204.
2. Obtain a copy of the remote service's WSDL contract.
3. Create a `ServiceFactory` instance using `ServiceFactory.newInstance()`.
4. Using the location of the remote service's WSDL contract and service name, create a new `Service` instance from the factory.
5. Using the QName of the `port` element defining the service and the name of the operation to be invoked, create a `Call` instance from the service.
6. Create the input parameters required to invoke the operation and store them in an `Object[]`.

Note: Only in and inout parameters are included in the `Object[]` used to invoke on the service. Do not include out parameters.

7. Invoke the remote service using the `Call` instance's `invoke()` method.

   Note: For oneway operations you can use `invokeOneWay()`.

8. Unpack any output parameters from the operation using the `Call` instance's `getOutputParameters()` method.

   Note: `getOutputParameters()` can return either a `Map` or a `List`.

Printer Service Demo

Overview

One use of dynamic invocations is in situations where you cannot be sure or the exact requirements of an operation. This can occur when a service may be fulfilled by a number of service providers. Each service provider may provide a service, such as document printing, but may have different operation signatures and require different information to fulfill the service request.

The application outlined below asks a service repository for an available printing service. The service repository can return two types of printing service: `Laser` and `InkJet`. The `print()` operation supported by a `Laser` printing service takes three arguments:

`Byte[] dataBuff`	The data to be printed.
`boolean duplex`	Specifies whether to use double sided printing.
`long numPage`	Specifies the number of pages to print per side.

The `print()` operation supported by an `InkJet` printing service takes two arguments:

`Byte[] dataBuff`	The data to be printed.
`boolean draft`	Specifies the print quality.

Both printing services return a cost for the printing. They also have one output parameter, `numSheets`, that specifies the number of sheets used to print the job.

The application uses the `Call` interface to invoke on the returned printing service. For purposes of demonstrating the use of the Call interface, the application is designed to not need to parse the returned WSDL contract to determine how to construct the invocation.

Application code

Example 280 shows the code for creating a print request and invoking on the returned print service.

Example 280: *Dynamic Invocation using the Call Interface*

```
//Java

import javax.xml.rpc.*;
import java.net.*
import com.iona.webservices.reflect.types.*;

Object[] args = null;
1 Bus bus = Bus.init();

2 QName name = new QName("http://www.printers.com",
                         "RegisteryService");
String portName = "RegisteryPort";
String wsdlPath = "file:/./printresistery.wsdl";
URL wsdlURL = new File(wsdlPath).toURL();
Register printReg = (Register)bus.createClient(wsdlURL, name,
                                               portName,
                                               Registery.class);

3 String printerType;
URIHolder tempURL;
QNameHolder tempName = new QNameHolder();
printReg.getPrinter(printerType, tempURL, tempName);

URL printerURL = tempURL.value.toURL();
QName printerName = tempName.value;
4 if (printerType.equals("Laser"))
{
  boolean duplex = true;
  long numPages = 2;
  // byte[] dataBuff obtained earlier
  args = new Object[]{dataBuff, duplex, numPages};
}
5 else if (printerType.equals("InkJet"))
{
  boolean draft = false;
  // byte[] dataBuff obtained earlier
  args = new Object[]{dataBuff, draft};
}
else System.exit(1);

6 ServiceFactory factory = ServiceFactory.newInstance();
```

Example 280:*Dynamic Invocation using the Call Interface*

```
7  Service printService = factory.createService(printerURL,
                                                 printerName);

8  String portName = name.getLocalPart().concat("Port");
   QName port = new QName("", portName);
9  Call printCall = printService.createCall(port, "print");

10 float cost = printCall.invoke(args);

11 Map outs = printCall.getOutputParameters();
12 long numSheets = outs.get("numSheets");

   System.out.println("Your print job costs "+cost+" and used "+
                      numSheets+" sheets of paper.");
```

What does the code do?

The code in Example 280 does the following:

1. Initialize the Artix bus.
2. Create a proxy for the print service registry.
3. Request a printing service from the print service registry.
4. If the type of printing service returned is `Laser`, build the three argument list.
5. If the type of printing service returned is `inkJet`, build the two argument list.
6. Get a new `ServiceFactory`.
7. Using the WSDL location and the service name returned from the print service registry, create a new `Service`.
8. Build the QName for the port defining the print service's endpoint.
9. Using the port name and the operation name, print, create a `Call`.
10. Invoke the print request using the argument list created above.
11. Get the output parameters as a `Map`.
12. Extract `numSheets` from the `Map`.

CHAPTER 18

Developing Plug-Ins

Plug-Ins can perform a number of tasks including registering servants or implementing handlers.

Overview

Developing and loading an Artix plug-in requires you to perform three tasks:

1. Extend the `BusPlugIn` class to implement your plug-in's application logic.
2. Implement the `BusPlugInFactory` interface.
3. Configure Artix to use the plug-in.

In this chapter

This chapter discusses the following topics:

Generating Plug-in Starting Point Code

Overview

Using the `wsdltojava` tool you can generate the code needed to implement a service defined by an Artix contract as a plug-in. In addition to the skeleton code generated for the service, `wsldtojava` will also generate a plug-in class that registers your servant and a plug-in factory to instantiate your plug-in.

Using wsdltojava

If you are planning on implementing a service as an Artix plug-in, you can use `wsdltojava` with the `-plugin` flag to generate the additional plug-in classes needed. The `-plugin` flag instructs the code generator to generate the following additional classes:

- *`portTypeName`*`ServicePlugin` extends `BusPlugIn` and includes code to register the appropriate servant with the bus in `busInit()`.
- *`portTypeName`*`ServicePluginFactory` extends `BusPlugInFactory` and includes the code to instantiate the generated plug-in class for your service.

In addition, the generated server mainline, *`portType`*`NameServer.java`, only includes code for initializing and starting the Artix bus. It contains comments about how to configure the service to load the plug-in at start up.

When not to use this

`wsdltojava -plugin` only works when you have a service defined in a contract. For this reason it is not usefull for building plug-ins that load handlers or implement a transport. Neither handlers or transports are defined in a contract.

You could use plug-in starting code generated for a service as a starting point, but you would have to completely rewrite the body of `busInit()` in the generated plug-in class. You would also have to modify the code that instantiated the plug-in in the generated plug-in factory.

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 281 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 281: *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 handler, you would register your handler factory in `busInit()`.

Example 282 shows a `busInit()` method used in implementing the bank service as a plug-in.

Example 282: *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 283.

Example 283: *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 284 shows the signature for `createBusPlugIn()`.

Example 284: *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 285 shows the signature for `destroyBusPlugIn()`.

Example 285: *destroyBusPlugIn()*

```
public void destroyBusPlugIn(BusPlugIn plugin);
```

Example

For example, the `BusPlugInFactory` implementation for a plug-in `BankPlugIn` would look similar to Example 286.

Example 286: *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
  {
  }
}
```

Configuring Artix to Load a Plug-in

Overview

All Java based plug-in have some common configuration entries that are required so that the bus can load the plug-in. These entries include:

- specifying the plug-in's factory class.
- loading the Java plug-in loader.
- adding the plug-in to the list of Java plug-ins to load.

In addition, there is an optional variable that specifies the classloader environment, if any, used by the plug-in.

Specifying a plug-in's factory class

To load a plug-in the bus needs to know which factory class is used to create instances of the plug-in's implementation. You specify the name of a plug-in's factory class using the variable `plugins:`*`plugin_name`*`:classname`. It takes a single string that is the name of the plug-in's factory class. You can place this variable in either an application specific scope or in the global scope. It is often better to place it in the global scope so that all applications in the configuration domain have access to the information.

Note: The name you give the plug-in in this variable must match the name you intend to use when listing the plug-in in the list of Java plug-ins to be loaded.

For example, if you created a plug-in to filter junk messages and called its factory class `JunkPluginFactory`, you would add the configuration line shown in Example 287 to the global scope of your Artix configuration file. When configuring an application to load this plug-in, you would refer to it as `junk`.

Example 287:*Configuring a Plug-in Factory Class*

```
plugins:junk:classname="JunkPluginFactory";
```

Loading the Java plug-in loader

Java plug-ins require that a special Java plug-in loader be used by the bus. You need to add this plug-in loader to the `orb_plugins` list of any application that uses Java plug-ins as shown in Example 288.

Example 288: *The Java Plug-in Loader in orb_plugins*

```
orb_plugins=[..., "java"];
```

Listing the Java plug-ins to be loaded by an application

Unlike C++ plug-ins which are listed in an application's `orb_plugins` list, Artix Java plug-ins are listed in a separate configuration variable called `java_plugins`. `java_plugins` is a list of comma separated plug-in names. The plug-in names used in the list must correspond to the name given the plug-in when specifying its factory class. For example to load the junk message plug-in configured in Example 287, you would use the configuration fragment shown in Example 289.

Example 289: *Loading a Java Plug-in*

```
orb_plugins=["java"];
java_plugins=["junk"];
```

Specifying a classloading environment

If you want your plug-in to use an Artix classloader environment, you specify the classloading environment using the `plugins:`*`plugin_name`*`:CE_Name` variable. The CE name is specified as a unique string.

In addition, you need to specify the location of the XML file describing the classloader environment. This is done with the `ce:`*`ce_name`*`:FileName` variable. *`ce_name`* is the CE name used when configuring the plug-in.

Example 290 shows a configuration fragment for loading the junk message plug-in using a classloader environment.

Example 290: *Using a Classloader Environment*

```
plugins:junk:CE_Name="junk_ce";
ce:junk_ce:FileName="\artix_ces\junk_ce.xml";
```

For more information on using classloaders see "Using Artix Classloader Environments" on page 573.

CHAPTER 19

Writing 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:

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 26.

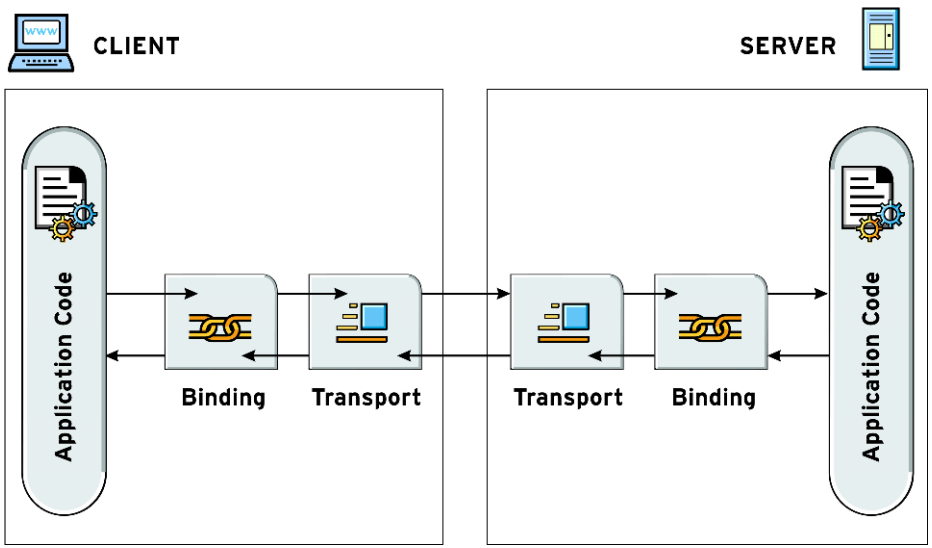

Figure 26: *The Life of a Message*

You can write 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 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 handler that takes the message from the transport and decompresses it before passing it on to the binding.

Handler levels

The JAX-RPC specification outlines a mechanism for developers to write custom 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 27.

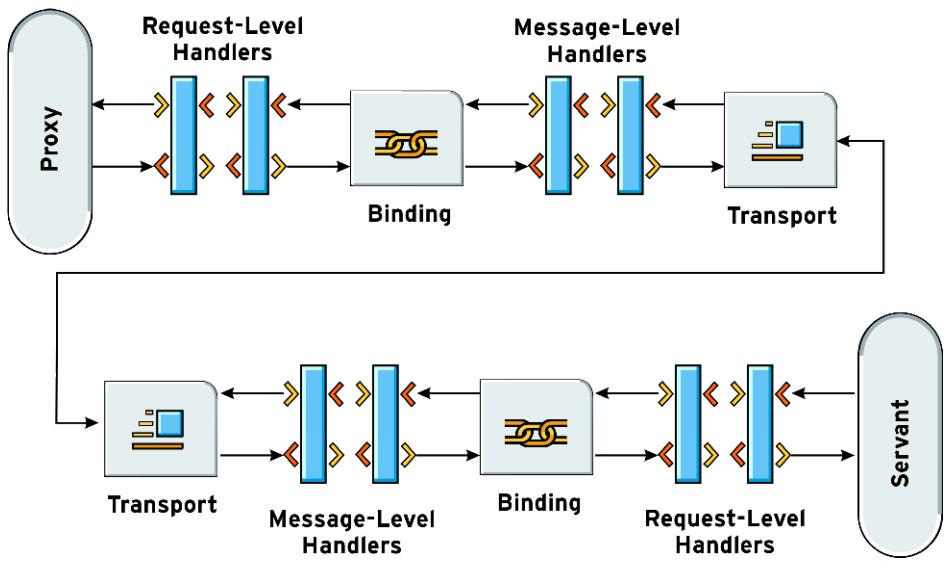

Figure 27: *Handler Levels*

On the client side of an application, you can write 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 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 handler

Handlers are developed as Artix plug-ins. This allows you to develop a handler once and 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 handlers. For details on the plug-in interfaces see "Developing Plug-Ins" on page 469.

To write a 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 handler.

Your `Handler` implementation contains the logic for the 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 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 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-level handler. Each method in a factory can instantiate one handler. However, a factory can be developed to

instantiate four handlers because the bus will only call the factory method needed to instantiate the handler configured to be used at a particular point in the message chain.

Configuring Artix to use handlers

Before your applications can use handlers, you must configure them to load the 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 handler names specifying the message-level handlers for a client.

binding:artix:client_request_interceptor_list is an ordered list of handler names specifying the request-level handlers for a client.

binding:artix:server_message_interceptor_list is an ordered list of handler names specifying the message-level handlers for a server.

binding:artix:server_request_interceptor_list is an ordered list of handler names specifying the request-level handlers for a server.

Note: A handler's name is the `String` used in the creation of the handler factory object.

The 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 `"Freeze+Dry"`, a message would be passed into the 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.

Example

Example 291 shows the configuration for an application that uses both client and server handlers.

Example 291:*Configuration with Handlers*

```
java_interceptors
{
  plugins:first_hand:classname="FirstHandlerPlugInFactory";
  plugins:second_hand:classname="SecondhandlerPlugInFactory";
  java_plugins = ["first_handler", "second_hand"];
  orb_plugins = ["xmlfile_log_stream","java"];

  client
  {
    binding:artix:client_request_interceptor_list =
   "firstHand+secondHand";
    binding:artix:client_message_interceptor_list =
   "firstHand+secondHand";

    # override config settings for client here
  };

  server
  {
    binding:artix:server_request_interceptor_list=
   "secondHand+firstHand";
    binding:artix:server_message_interceptor_list =
   "secondHand+firstHand";

    # override config settings for server here
    };
};
```

Creating the Handler Plug-in

Overview

Artix handlers need to be hosted in a plug-in. Creating a plug-in for your handlers follows the same pattern as creating any other Java plug-in. The difference is that in `BusPlugin.busInit()` you register the handler factories used to instantiate your handlers.

Procedure

To create a plug-in for your handlers do the following:

1. Implement a `BusPluginFactory` to load the plug-in that implements your handler. See "Implementing the BusPlugInFactory Interface" on page 474.
2. Extend `BusPlugin` to load your handler using the bus' `registerHandlerFactory()` method.

If you wish to have a single plug-in load multiple handlers, make multiple calls to `registerHandlerFactory()`.

The plug-in

The implementation of `busInit()` in your plug-in registers the handler factories for the handlers used by the application. Handler factory registration is done using the bus' `registerHandlerFactory()` method. The signature for `registerHandlerFactory()` is shown in Example 292.

Example 292: *registerHandlerFactory()*

```
void registerHandlerFactory(HandlerFactory factory);
```

`registerHandlerFactory()` takes an instance of the handler factory for your 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

Example 293 shows a the plug-in code for a handler.

Example 293: *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 firstHandFactory());
       bus.registerHandlerFactory(new secondHandFactory2());
     }
     catch (Exception ex)
     {
       throw new BusException(ex);
     }
   }

6   public void busShutdown() throws BusException
   {
   }
}
```

The code in Example 293 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 handlers' factories with the bus using `registerHandlerFactory()`.

6. Implements `busShutdown()`.

Creating a Handler Factory

Overview

The bus calls the methods provided by the `HandlerFactory` you register in the handler plug-in. You need to implement a `HandlerFactory` for each set of handlers you need. The `HandlerFactory` interface has four methods:

- `getClientRequestHandler()` creates a client-side, request-level handler.
- `getServerRequestHandler()` creates a server-side, request-level handler.
- `getClientMessageHandler()` creates a client-side, message-level handler.
- `getServerMessageHandler()` creates a server-side, message-level handler.

If all four methods are implemented, one `HandlerFactory` can instantiate one of each type of handler.

The GenericHandlerFactory

The easiest way to develop your handler factory is to extend the `GenericHandlerFactory` included with Artix. The `GenericHandlerFactory` implements all of the methods in the `HandlerFactory` interface. You only need to override the methods needed for your handlers and provide a constructor for your handler factory.

Implementing the methods

When using the `GenericHandlerFactory` as a base class, you only need to implement the methods that relate to your application. For example if your application only uses a server-side, message-level handler, you only need to implement `getServerMessageHandler()`. If, however, your application also uses a client-side. message-level handler, you will also need to implement `getClientMessgeHandler()`.

The signatures for the `HandlerFactory` methods are shown in Example 294. They take a single `HandlerInfo` object and return an instance of the class `HandlerInfo`.

Example 294: *Handler Factory Methods*

```
public HandlerInfo getClientRequestHandler(HandlerInfo info)
public HandlerInfo getServerRequestHandler(HandlerInfo info)
public HandlerInfo getClientMessageHandler(HandlerInfo info)
public HandlerInfo getServerMessageHandler(HandlerInfo info)
```

The factory methods need to supply the `Class` that implements your handler. For example if your client-side handler is implemented by a class called `firstHandRequestHandler`, you need to set the returned HandlerInfo's `HanderClass` field to `firstHandClientRequestHandler.class` by invoking `setHandlerClass()` on the `HandlerInfo` object.

Example

Example 295 shows code for implementing a handler factory.

Example 295: *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 firstHandFactory extends GenericHandlerFactory
   {
2    public fristHandFactory()
     {
       super(new String("firstHand"));
     }

3    public HandlerInfo getClientRequestHandler(HandlerInfo info)
     {
4      info.setHandlerClass(firstHandClientRequestHandler.class);
       return info;
     }
```

Example 295:*Handler Factory For Request Level Handlers*

```
  public HandlerInfo getServerRequestHandler(HandlerInfo)
  {
    info.setHandlerClass(secondHandServerRequestHandler.class);
    return info;
  }
}
```

The code in Example 295 does the following:

1. Extends `GenericHandlerFactory`.
2. Implements a constructor for the handler factory. The string set is the string used by the bus to reference the handler factory. It is also the value which is used in the configuration file to refer to the handler factory.
3. Overrides `getClientRequestHandler()`.
4. Sets the `HandlerClass` property to the class of the handler that will process client requests.

HandlerInfo

The `HandlerInfo` passed into the method contains the following information:

- The current bus
- The QName of the service for which the handler is being created
- The name of the port for which the handler is being created

To retrieve this information you first need to get the configuration map from the HandlerInfo object as shown in Example 296.

Example 296:*Getting a Configuration Map from a HandlerInfo*

```
import java.util.Map;

Map config = info.getHandlerConfig();
```

To access the properties stored in the configuration map use the Artix handler constants shown in Table 30.

Table 30: *Configuration Map Properties*

Property	Description
`HandlerContants.BUS`	Returns the current bus.
`HandlerConstants.SERVICE_NAME`	Returns the QName of the service for which the handler is being created.
`HandlerConstants.PORT_NAME`	Returns the name of the port through which messages for this handler will pass.

Example 297 shows code for getting all of the properties from a `HandlerInfo` object.

Example 297:*Getting Configuration Information From a HandlerInfo*

```
import java.util.Map;
import com.iona.jbus.*;
import com.iona.jbus.HandlerConstants;

Map config = info.getHandlerConfig();
Bus bus = (Bus)config.get(HandlerConstants.BUS);
QName serv = (QName)config.get(HandlerConstants.SERVICE_NAME);
String port = (String)config.get(HandlerConstants.PORT_NAME);
```

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, handlers at the request level can access the following pieces of data:

- The name of the invoked operation
- The parameters of the invoked operation
- The application's message context
- Any Artix-specific context information that is set using the `IonaMessageContext`
- The message's SOAP headers
- The message's security properties

For example, your application could have a client side handler that added a custom SOAP header to its requests for authorization purposes. The server could then use a 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 `BusPlugin` to load your handler. See "Creating the Handler Plug-in" on page 485.
2. Implement the request-level handler methods in your `HandlerFactory` so the bus can instantiate your handler. See "Creating a Handler Factory" on page 488.
3. Implement a `Handler` to host the logic used by your handler.
4. Configure your application to load the handler plug-in.

5. Configure your application to include the handler in the request handler chain. See Deploying and Managing Artix Solutions.

The handler implementation

The easiest way to develop your handler logic is to extend the `com.iona.jbus.jaxrpc.handlers.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 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 handler is called. For example, a 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 298. Both methods have a `MessageContext` as an argument. For information on using the message contexts see "Using Message Contexts" on page 267. The return value should reflect the state of the message processing. If the message is successfully processed return `true`. If not, return `false`.

Example 298: *handleRequest() and handleResponse()*

```
boolean handleRequest(MessageContext context);
boolean handleResponse(MessageContext context);
```

At the request-level, your handler can access the generic message context or 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 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-level 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

Example 299 shows the code for a client request-level handler that sets a SOAP header on the request and reads the SOAP header returned with the response.

Example 299: *Client Request Level Handler*

```
// Java
import com.iona.jbus.IonaMessageContext;
import com.iona.jbus.ContextException;
com.iona.jbus.jaxrpc.handlers.GenericHandler;

import javax.xml.namespace.QName;

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;
  }

  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.

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 handler that compresses the message data to enhance network performance. The server could then use a handler to decompress the message data before it is sent to the binding for decoding.

Procedure

To develop a message-level handler you need to do the following:

1. Implement a `BusPlugin` to load your handlers. See "Creating the Handler Plug-in" on page 485.
2. Implement the message-level methods in your `HandlerFactory` so the bus can instantiate your handler. See "Creating a Handler Factory" on page 488.
3. Implement a `Handler` to host the logic used by your handler.
4. Configure your application to load the handler plug-in.
5. Configure your application to include the handler in the handler chain. See Deploying and Managing Artix Solutions.

The handler implementation

The easiest way to develop your 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 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 handler is called. For example, a 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 300. Both methods have a `MessageContext` as an argument. For information on using the message contexts see "Using Message Contexts" on page 267. The return value should reflect the state of the message processing. If the message is successfully processed return `true`. If not, return `false`.

Example 300:*handleRequest() and handleResponse()*

```
boolean handleRequest(MessageContext context);
boolean handleResponse(MessageContext context);
```

At the message level, your handler can access both the generic message context and a special `StreamMessageContext` that provides access to the raw message data that is to be written onto the wire. For more information on using the stream message context, see "Manipulating Messages as a Binary Stream" on page 516. In addition, if you are using the SOAP binding, you can access the SOAP message context. For more information on working with the SOAP message context, see "Working with SOAP Messages" on page 513. 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 either the raw message data or the SOAP message context.

Example

Example 301 shows the code for a client message-level 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.

Example 301:*Client Message-Level Handler*

```
// Java
import com.iona.jbus.*;
com.iona.jbus.jaxrpc.handlers.GenericHandler;

import java.io.*;
import javax.xml.namespace.QName;

public class firstHandClientMessageHandler 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;
  }
}
```

Handling Errors and Exceptions

Overview

Java handlers have three ways of generating errors when processing a message:

- throw a runtime exception.
- throw a user-exception that is wrapped in a runtime exception.
- populate the message context with an error message and return false.

The behavior of the handler depends on if the message being processed is a request or a response. The resulting behavior also depends on if the handler is implemented on the client-side or the server-side of an application.

In this section

This section discusses the following topics:

Handling Errors when Processing Requests

Overview

As requests are passed down the messaging chain, they are processed by each handler's `handleRequest()` method. Regardless of where on the messaging chain a request is, an error will prevent the request from making it to the service implementation.

Client-side

If an exception is thrown at any point in the client's request processing chain, it is returned immediately to the client. All handlers in the messaging chain are skipped and no message processing is done.

If `handleRequest()` returns `false`, the handler is responsible for populating the response buffer with an appropriate fault message. Artix then invokes the handler's response chain starting from the handler that created the fault condition. The fault message will be processed as if it were a normal response and each handler's `handleResponse()` method will process it.

Server-side

Error processing on the server-side is more complicated. The behavior of the service depends on where in the messaging chain the error condition is encountered.

At the message-level, throwing an exception will cause the messaging chain to stop processing the message. The bus will the create a fault message containing the exception, place it in the response buffer, and return the fault to the client. The response message is passed back down the handler chain and precessed by each message handler's `handleFault()` method.

If a message-level request handler returns `false`, you must ensure that an appropriate response message is created and placed in the response buffer. A return of `false` from a message-level request handler will cause the bus to stop processing the request and return the message in the response buffer to the client. The response handler sequence is followed starting from the handler that created the error condition. The messages are processed through the `handleResponse()` method.

At the request-level, throwing an exception will cause the messaging chain to stop processing the message. The bus will then send the response back down the message chain starting from the handler that generated the

exception. However, instead of calling `handleResponse()` on each handler, the bus will call `handleFault()`. In this instance, the servant will never be invoked.

Returning `false` will cause the messaging chain to stop processing the request and forward the request straight to the servant for processing.

Handling Errors when Processing Responses

Overview

As responses are passed down the messaging chain, they are processed by each handler's `handleResponse()` method. At this point in the request/response chain, it is expected that the response buffer is already populated. However, the contents of the request buffer is not fixed.

Server-side

On the server-side, request-level handlers can safely throw runtime exceptions. The exception will stop the further processing of handlers along the server's message chain. The exception will be immediately sent to the client as a fault message. As the fault message is passed back down the message handler chain it is processed by each handler's `handleFault()` method.

At the message-level, throwing an exception will cause the messaging chain to stop processing the message. The bus will the create a fault message containing the exception, place it in the response buffer, and return the fault to the client. The response message is passed back down the handler chain and precessed by each message handler's `handleFault()` method.

Server-side response handlers that return `false`, at both the request-level and the message-level, have no effect on message processing. Regardless of the return value from `handleResponse()`, the server will continue to send the message along the messaging chain. The message will pass through all of the handlers in the chain.

Throwing User Faults

Overview

In cases where you want to pass a user defined exception back to the client application, you can wrap the user defined exception in a runtime exception and send it back to the client. Artix will catch the runtime exception and inspect its contents. If the runtime exception contains a user defined fault, then Artix passes the user defined fault up the messaging chain. If not, Artix just passes the runtime exception up the messaging chain.

Procedure

To throw a user defined fault from a message handler do the following:

1. Ensure that your service definition, in the service's contract, includes a fault message. See "Describing User-defined Exceptions in an Artix Contract" on page 160.
2. Create an instance of the user defined fault you want to throw. See "Working with User-defined Exceptions in Artix Applications" on page 165.
3. Throw a `RuntimeException` using the created instance of your user defined fault as the parameter to the constructor.

When the Artix client transport layer receives the exception it will discover that it contains a user defined exception, remove it from the `RuntimeException` wrapper, and pass the user defined exception up the messaging chain. As the message is passed up the messaging chain it will be processed by the `handleFault()` method of the message handlers.

Example

If you had a service that could return a user defined fault called `Pied` its contract would contain a fragment similar to Example 302.

Example 302: *Service Definition with a Fault*

```
...
<message name="pied">
  <part name="flavor" type="xsd:string" />
</message>
```

Example 302:*Service Definition with a Fault*

```
...
<portType name="brainService">
  <operation name="tonight">
    <input message="tns:marketData" name="plan" />
    <output message="tns:worldDominiation" name="goal" />
    <fault message="tns:pied" name="pinky" />
  </operation>
</portType>
```

The contract fragment in Example 302 would cause Artix to generate a Java class called `Pied` that extended the class `Exception`. `Pied` would contain a single member variable called `flavor`. Because `Pied` extends `Exception`, it inherits from `Throwable` which means it can be used as an argument the `RuntimeException` object's constructor.

If you wanted to throw a pied exception from a message handler, you would use code similar to Example 303.

Example 303:*Throwing a User Defined Exception in a MessageHandler*

```
public class cageBreak extends GenericHandler
{
  public boolean handleRequest(MessageContext context)
  {
   ...
    Pied userFault = new Pied("bananaCream");
    throw RemoteException(userFault);
    ...
    return true;
  }

...
}
```

Processing Fault Messages

Overview

Fault messages are processed by the `handleFault()` method of a handler. It is implemented in the same manner as the other message handler functions.

Implementing the fault handler

Like `handleRequest()` and `handleResponse()`, `handleFault()` receives a generic `MessageContext` as a parameter. Its signature is shown in Example 304.

Example 304: *handleFault()*

```
public boolean handleFault(MessageContext context)
```

The information available from the `MessageContext` depends on where in the messaging chain the handler is placed. At the request-level, the fault handler can access any information in the generic `MessageContext` and any information in the `IONAMessageContext`. For information on using the `IONAMessageContext`, see "Using Message Contexts" on page 267.

At the message-level, the fault handler can access the `SOAPMessageContext`, if the service uses a SOAP payload format, or the `StreamMessageContext`. For information on using the `SOAPMessageContext` or the `StreamMessageContext`, see "Manipulating Messages in a Handler" on page 507.

Reading the contents of the exception

Server-side request-level message handlers can access the contents of an exception thrown by the servant in `handleFault()` in much the same way that they access the information about an operation in `handleResponse()`. You call the `getProperties()` method on the context using `ContextConstants.SERVER_RESPONSE_EXCEPTION` as the property name. The property is returned as a generic Java object that needs to be cast into either the actual class of the specific exception or one of the generic subclasses used to create the exception.

Example 305 shows code for getting an exception in `handleFault()`.

Example 305: *Accessing an Exception*

```
handleFault(MessageContext context)
{
  Throwable ex = (Throwable)context.getProperty(ContextConstants.SERVER_RESPONSE_EXCEPTION);

  //process the exception
  ...
}
```

Return values

`handleFault()` returns a boolean value. If `handleFault()` returns `true`, the message continues along the massaging chain as normal. If `handleFault()` returns `false`, the bus stops processing the message and returns it directly to the client. In the case where `handleFault()` returns `false`, it is the handler's responsibility to ensure that the response message contains an appropriate message.

Throwing exceptions

If `handleFault()` throws an exception, the exception is returned directly to the client. If the exception is thrown while in the server-side messaging chain, the client-side messaging chain will process the returned fault message normally. If the exception is thrown while in the client-side messaging chain, the exception is immediately returned to the user code.

CHAPTER 20

Manipulating Messages in a Handler

One function of a handler may be to modify messages as they pass between the application level code and the wire.

Overview

Handlers often need to have a fine grained access to the messages they process. Artix provides access to the message details in the handlers in several ways. Request-level handlers can access the parameters passed as part of an operation invocation. Message-level handlers can access the message information as raw stream data using the `StreamMessageContext`. In addition, if your application uses a SOAP binding, your message-level handlers can also access message data using the JAXM SOAP APIs through the `SOAPMessageContext`.

In this chapter

This chapter discusses the following topics:

Working with Operation Parameters

Overview

Request-level handlers in Artix have access to the name of the operation which generated the message and the message parts, which represent the operation parameters, of both the request message and the response message. You can use this information to determine how a message is to be processed. You can also change the values of the message parts as they are passed along the message chain.

Getting the operation name

You get the name of the operation from which the message being processed originated through the generic message context. It is stored in a property accessed using the Artix constant `ContextConstants.OPERATION_NAME`. The returned value is a `String` containing the operation name as listed in the Artix contract.

WARNING: Changing this value can produce unpredictable results.

For example, if you have a contract with the interface defined in Example 306 the operation name returned from the context would be `forward`.

Example 306: *Example Port Type*

```
<message name="travelRequest">
  <part name="date" type="xsd:string"/>
</message>
<message name="travelResponse">
  <part name="arrived" type="xsd:boolean"/>
</message>
<portType name="tardis">
  <operation name="forward">
    <input message="travelRequest" name="request"/>
    <output message="travelResponse" name="outcome"/>
  </operation>
</portType>
```

Example 307 shows the code for getting the operation name from the message context.

Example 307: *Getting the Operation Name*

```
import com.iona.jbus.ContextConstants;

public class ServerRequestHandler extends GenericHandler
{
  public boolean handleRequest(MessageContext context)
  {
    String opName = (String)
   context.getProperty(ContextConstants.OPERATION_NAME);
   ...
  }
...
}
```

Message part context properties

Artix uses four separate context properties for storing message parts:

- `CLIENT_REQUEST_VALUES` holds the message parts for an outbound request on the client-side of the messaging chain.
- `SERVER_REQUEST_VALUES` holds the message parts for an inbound request on the server-side of the messaging chain.
- `SERVER_RESPONSE_VALUES` holds the message parts for an outbound response on the server-side.
- `CLIENT_RESPONSE_VALUES` holds the message parts for an inbound response on the client-side.

The values are stored as an array of generic Java `Object` objects that can be cast back into their proper types for manipulation. The returned array contains values for all parts in the message that are set. If a message part is `nillable`, it will not be included in the returned array if was not populated.

In addition to storing message parts, Artix also stores a list of each parts Java class. This list is an array of `Class` objects and it contains information on all of the possible parts in a message. There are also four context properties for storing the message parts' class list:

- `CLIENT_REQUEST_CLASSES` holds the class information for the message parts of an outbound request on the client-side of the messaging chain.
- `SERVER_REQUEST_CLASSES` holds the class information for the message parts of an inbound request on the server-side of the messaging chain.

- SERVER_RESPONSE_CLASSES holds the class information for the message parts of an outbound response on the server-side.
- CLIENT_RESPONSE_CLASSES holds the class information for the message parts of an inbound response on the client-side.

Accessing the message parts

You can access the parts of a message using the `getProperties()` method on the generic message context in request-level handlers. While, you can pass in any of the message part property identifiers into `getProperties()`, only the message parts appropriate to the position in the message chain have valid values. For example, if your handler is a server-side response handler, only the properties SERVER_RESPONSE_CLASSES and SERVER_RESPONSE_VALUES have data. If you try to access any of the other message part properties, `getProperties()` will return `NULL`.

Working with the message parts

Artix returns the message parts as an array of Java `Object` objects when you request the message part values. The returned array contains all of the non-nill message parts. If a message part is `nillable` and not set, there will not be a place holder in the returned array of objects.

To inspect or change any of the message parts, you can cast it to the appropriate type and work with it as you would normally. All changes made to the value of a message part are immediately reflected in the message.

The only restriction to manipulating message parts in Java handlers is that you cannot add or remove a message parts. This also means that you cannot change the value of a nill message part.

Working with message part class information

Artix returns message part class information as an array of `Class` objects. The returned array has an entry for every part specified in the WSDL description of the message. If a message part is nillable and not set by the operation, the message part's class information will still be returned.

You should not change any of the values in the returned array. It is only stored for information purposes. For instance you could compare the list of parts to the list of classes to determine if a message part is not set.

Example

If you were developing an ordering system for kayak paddles for a manufacturer in Europe that takes orders from retailers in the United States, you may need to convert the paddle lengths from inches to centimeters. The interface for such an ordering system is shown in Example 308.

Example 308: *Paddle Ordering Interface*

```
<message name="order">
  <part name="amt" type="xsd:int" />
  <part name="length" type="xsd:int" />
</message>
<message name="bill">
  <part name="amtDue" type="xsd:float" />
</message>
<portType name="supplyPaddles">
  <operation name="orderPaddles">
    <input message="tns:order" name="order" />
    <output message="tns:bill" name="bill" />
  </operation>
</portType>
```

Example 309 shows s server-side request handler that converts the `length` part of an incoming request from inches to centimeters.

Example 309: *Changing the Value of Message Parts*

```
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.namespace.QName;
import com.iona.jbus.ContextConstants;

public class ServerRequestHandler extends GenericHandler
{
  public boolean handleRequest(MessageContext context)
  {
1   Object[] parts = (Object[])
    context.getProperty(ContextConstants.SERVER_REQUEST_VALUES);

2   int length = (int)parts[1];
3   parts[1] = length * 2.54;

4   return true;
  }
}
```

The code in Example 309 does the following:

1. Gets the server request message parts from the message context.
2. Gets the `length` part of the message. As shown in Example 308 on page 511, `length` is the second part in the request.
3. Converts the `length` part from inches to centimeters.
4. Returns `true` to continue message processing.

Working with SOAP Messages

Overview

Message-level handlers in Artix can, if they are used by application with a SOAP binding, access and modify the SOAP message being sent between the participating services. Using the `SOAPMessageContext` class, developers can get the message being passed as a `javax.xml.soap.SOAPMessage` object and manipulate the message using the standard Java APIs.

SOAPMessageContext

`SOAPMessageContext` extends the generic `MessageContext` class that is passed into all message handlers. It is only available in message-level handlers for applications that have a SOAP binding. If your application is not using a SOAP binding and you attempt to use the `SOAPMessageContext` you will get an exception.

`SOAPMessageContext` has two methods that allow you to retrieve and modify the contents of the SOAP message being processed by a handler. They are described in Table 31.

Table 31: *SOAPMessageContext Methods*

Signature	Description
`SOAPMessage getMessage()`	Returns the `SOAPMessage` contained in the context.
`void setMessage(SOAPMessage message)`	Sets the `SOAPMessage` contained in the context to the message specified.

SOAPMessage

Once you have the `SOAPMessageContext`, you can use it to manipulate the SOAP message using the `SOAPMessage` APIs. The `SOAPMessage` implementation in Artix conforms to the *SOAP with Attachments API for Java (SAA)J 1.2* specification. Using this API, you can access all parts of the SOAP message elements. These are listed in Table 32.

Table 32: *SOAPMessage Elements*

Element	Description
`SOAPPart`	Contains routing and identification information for the message. All `SOAPMessages` must have a valid `SOAPPart`.
`SOAPEnvelope`	Contained inside of the `SOAPPart`. By default, this object contains an empty `SOAPHeader` and an empty `SOAPBody`.
`SOAPBody`	Contains the data passed in the SOAP message. All data must be XML data.
`SOAPHeader`	An optional element of the SOAP message that contains XML data. This element provides a container for additional information such as security information.
`AttachmentPart`	Optional elements of a SOAP message that can contain binary data such as images or word processing documents.

For more information on the `SOAPMessage` APIs see the SAAJ 1.2 specification or the publicly available J2EE API documentation.

Example

Example 310 shows an example of using the `SOAPMessageContext` to add an attachment to a SOAP message.

Example 310: *Using the SOAPContext*

```
//Java
boolean handleRequest(MessageContext context)
{
1   SOAPMessageContext SOAPcontext = (SOAPMessageContext)context;

2   SOAPMessage message = SOAPcontext.getMessage();

3   Java.awt.Image image = getPicture();

4   AttachmentPart imagePart = message.createAttachmentPart(image,
                                                             "img/gif");
5   message.addAttachmentPart(imagePart);

6   message.saveChanges();

7   SOAPcontext.setMessage(message);
}
```

The code in Example 310 does the following:

1. Gets the `SOAPMessageContext` by casting the passed in `MessageContext`.
2. Gets the `SOAPMessage` stored in the context.
3. Gets the image to store in the SOAP message.

   Note: You are left to implement the `getPicture()` method.

4. Creates a new `AttachmentPart` to store the image.
5. Adds the new `AttachmentPart` to the message.
6. Updates the message's data.
7. Sets the modified message back into the `SOAPMessageContext`.

Manipulating Messages as a Binary Stream

Overview

While the `SOAPMessageContext` provides a more convenient means of accessing the contents of a message, it only works when the service is using a SOAP payload format. If your service does not use a SOAP payload format or you cannot be sure what payload format your service is going to use, you can access the contents of messages using the `StreamMessageContext`.

The `StreamMessageContext` returns the contents of a message as either a Java `InputStream` or a Java `OutputStream`. Using these binary streams, you can then manipulate the contents of the message as needed. It is important to remember, however, that the service receiving the message can accept the alterations made to the message.

Getting the StreamMessageContext

To get a `StreamMessageContext` you cast the `MessageContext` passed into the handler method as shown in Example 311.

Example 311:*Getting a StreamMessageContext*

```
// Java
boolean handleResponse(MessageContext context)
{
  StreamMessageContext myCtx = (StreamMessageContext)context;
  ...
}
```

Getting message streams

The `StreamMessageContext` has methods for getting and setting the input and output streams used by the transport as shown in Example 312. 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 312:*StreamMessageContext*

```
package com.iona.jbus;
```

Example 312:*StreamMessageContext*

```
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

Example 313 shows code for adding a string to the end of a message.

Example 313: *Using StreamMessageContext*

```
class TestInputStream extends InputStream
{
  InputStream in;
  ByteArrayInputStream bin;

  TestInputStream(InputStream i2, byte bytes[])
    {
      in = i2;
      bin = new ByteArrayInputStream(bytes);
    }

  public int read() throws IOException
  {
    if (bin != null)
    {
      int i = bin.read();
      if (i == -1) bin = null;
      else return i;
    }

    return in.read();
  }
}
...
boolean handleResponse(MessageContext context)
{
  String message = "San Dimas High School Football Rules!";
  byte bytes[] = message.getBytes();

  StreamMessageContext smc = (StreamMessageContext)context;
  InputStream ins = smc.getInputStream();
  ins = new TestInputStream(ins, bytes);
  smc.setInputStream(ins);
}
```

CHAPTER 21

Developing Custom Artix Transports

Artix provides a number of standard transport plug-ins. However, your applications my use a custom transport that is not provided. Using the Artix plug-in mechanism, developing custom transports in Java is a straightforward procedure.

In this chapter

This chapter discusses the following topics:

Developing a Transport: The Big Picture

Overview

All of the transports used by Artix are implemented as plug-ins that are loaded based on cues from an application's Artix contract. The implementation of transports in plug-ins makes it easy to develop custom Artix transports. This is useful in situations where you have applications that use a homegrown transport.

What does a transport do?

Artix transports are responsible for reading data from and writing data to an Artix endpoint. A transport first establishes a connection with the target endpoints and then waits to perform work. When reading data from the wire, a transport plug-in reads the raw binary data, decodes any transport specific header information, and passes the message to the binding as a binary buffer. When writing data to the wire, a transport plug-in receives a formatted message from the binding as a binary buffer, adds any transport specific headers, and sends the binary data to the target endpoint.

The transport WSDL definition

Every transport requires some piece of information from the user before it can connect two endpoints. In the simplest case, the only information needed is the address where messages are sent and received. More complex transports may require more information such as persistence and security settings. In all cases, this information is supplied in an application's Artix contract. Transport configuration is supplied inside the WSDL `port` element that defines an endpoint.

For each Transport used by Artix there is a corresponding XMLSchema document describing the WSDL extension element that defines the transport attributes. When designing a custom transport, you will also need to define the transport attributes in an XMLSchema document.

Procedure

To develop a custom Artix transport you need to do the following:

1. Make an XMLSchema document defining the attributes needed to define an endpoint for your transport.
2. Extend the `TransportFactory` class.
3. Implement an Artix plug-in that registers your transport factory.

4. Implement the `ClientTransport` interface as shown in "Developing the Client Transport" on page 535.
5. Implement the `ServerTransport` interface as shown in "Developing the Server Transport" on page 543.

Making a Schema for the Transport Attributes

Overview

Like most parts of Artix, transport endpoints are defined by an application's contract. The transports, other than SOAP/HTTP, are defined using an XMLSchema document that defines an extension to WSDL. When you create a custom transport you must also define the WSDL extensions for defining an endpoint for the newly developed transport. The XMLSchema document defining your transport's attributes will also be specify the namespace identifying your transport so that Artix can load it dynamically.

Transport namespace

The namespace you assign to a transport is important for two reasons. First it allows you to validate your endpoint definition against the XMLSchema you develop to define its WSDL extensions. Second, and more important, it informs Artix to load your transport at runtime. When Artix parses an application's contract it decides what transport and binding plug-ins to load based on the namespaces used in the contract's `port` elements and their corresponding `xmlns` entries in the contract's `definition` element.

For example, when using the Artix IIOP tunnel transport you include `xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"` in the contract's `definition` element. When defining the endpoint you use the `service` element shown in Example 314.

Example 314: *Endpoint Definition*

```
<service name="IIOPservice">
  <port name="IIOPport" binding="tns:IIOPbinding">
    <iiop:address location="file:///objref.ior" />
    <iiop:policy persistent="true" />
  </port>
</service>
```

When parsing the `port` element, Artix would resolve the `iiop` tag to the namespace specified in the `definition` element and then know to load the IIOP tunnel transport plug-in. For more information on how to specify the configuration for a transport see, "Using your Custom Transport" on page 560.

When writing the XMLSchema for your transport's attributes you specify the transport's namespace as the target namespace. This is done using the `targetNamespace` attribute of the XMLSchema document's `schema` element, as shown in Example 315.

Example 315: *Specifying the Transport's Namespace*

```
<xs:schema
  targetNamespace="http://widgetVendor.com/transport/socket"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:sock="http://widgetVendor.com/transport/socket"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
```

When defining an endpoint that uses the transport defined with the statement in Example 315, your contract needs to include `xmlns:sock="http://widgetVendor.com/trasnport/socket"` in its `definition` element. The `port` element defining the endpoint's attributes would contain elements prefixed `sock` to specify that they used the custom transport.

Defining the transport attributes

Transport attributes are defined as WSDL extensibility elements according to the WSDL 1.1 specification. To properly define your transport's attributes as WSDL extensions your XMLSchema definition must conform to the following rules:

1. It must import the WSDL 1.1 XMLSchema document defined in the namespace `http://schemas.xmlsoap.org/wsdl/`.
2. All the elements that define attributes to be listed in the Artix contract must be of a type that extends the abstract `wsdl:tExtensibilityElement` type.

Beyond these two restrictions your transport's attributes can be as complex or as simple as needed to fully define an endpoint. For example, the IIOP tunnel transport has a single required element to specify the endpoint's address. However, the MQ transport has two elements each of which can take a number of attributes to define an endpoint.

Example

Example 316 shows an example of an XMLSchema document for a transport that uses a single element, `sock:address`, to define an endpoint.

Example 316: *Sample Transport XMLSchema*

```
<xsd:schema
  targetNamespace="http://widgetVendor.com/transport/socket"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sock="http://widgetVendor.com/transport/socket"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/"/>
  <xsd:complexType name="addressType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:tExtensibilityElement">
        <xsd:attribute name="host" type="xsd:string"
                       use="required">
        <xsd:attribute name="port" type="xsd:string"
                       use="required">
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="address" type="sock:addressType"/>
</xsd:schema>
```

Example 316 does the following:

1. Defines the target namespace for the transport's attributes.
2. Imports the WSDL XMLSchema definition.
3. Defines a complex type, `addressType`, that extends `wsdl:tExtensibilityElement` and has one required attribute, `location`.
4. Defines the element `address`.

When you wanted to define an endpoint for the transport defined in Example 316 you would include `xmlns:sock="http://widgetVendor.com/transport/socket"` in the contract's `definition` element and a `service` element similar to Example 317.

Example 317:*Socket Endpoint Definition*

```
<service name="widgetSocketService">
  <port name="widgetSocketPort> binding="tns:widgetSOAPbinding">
    <sock:address host="localhost" port="8090" />
  </port>
</service>
```

Developing and Registering the Transport Factory

Overview

Transports are created and managed by the bus, so each transport must have a transport factory. You create a transport factory by extending `TransportFactory`. The transport factory is responsible for creating any resources needed by the transport and setting the threading model used by the transport.

Transports are loaded by the Artix bus using the plug-in mechanism. So to use a transport you must write a plug-in that instantiates a transport factory for your transport. The plug-in must also register the transport factory with the bus. For a detailed discussion of implementing a plug-in see "Developing Plug-Ins" on page 469.

In this section

This section discusses the following topics:

Creating a Transport Factory

Overview

Transports are managed by the bus using a transport factory. The transport factory allows the bus to create transport instances, to initialize the transport with the desired policies, and to eventually shutdown the transport. You create a transport factory for your transport by extending the abstract `com.iona.jbus.TransportFactory` class.

TransportFactory methods

`TransportFactory` has six methods that must be implemented. These are explained in Table 33.

Table 33: *Method for Transport Factory*

Method	Function
`ClientTransport createClientTransport()`	This method is responsible for instantiating an instance of your `ClientTransport` implementation. In addition, you can initialize any resources needed by your client transport.
`void destroyClientTransport(ClientTransport transport)`	This method is responsible for cleaning up any resources used by your `ClientTransport` implementation.
`ThreadingModel getClientThreadingModel()`	This method is responsible for specifying the threading model used by your client transport. For details about the available threading models see "Transport threading models" on page 530.
`ServerTransport createServerTransport()`	This method is responsible for instantiating an instance of your `ServerTransport` implementation. In addition, you can initialize any resources needed by your server transport.
`void destroyServerTransport(ServerTransport transport)`	This method is responsible for cleaning up any resources used by your `ServerTransport` implementation.

Table 33: *Method for Transport Factory*

Method	Function
`ServerTransportPolicies getServerTransportPolicies()`	This method is responsible for specifying the threading model used by your server transport, who supplies threads to the transport, and if the transport can support concurrent requests. For details about the available threading models see "Transport Policies" on page 530.

Example

Example 318 shows a transport factory for a custom transport.

Example 318: *SocketTransportFactory*

```
import com.iona.jbus.*;

public class SocketTransportFactory extends TransportFactory
{
  private final ServerTransportPolicies serverPolicies = new DemoServerTransportPolicies();

  public ClientTransport createClientTransport()
  {
    return new SocketClientTransport();
  }

  public void destroyClientTransport(ClientTransport transport)
  {
  }

  public ThreadingModel getClientThreadingModel()
  {
    return ThreadingModel.MULTI_THREADED;
  }

  public ServerTransport createServerTransport()
  {
    return new SocketServerTransport();
  }

  public void destroyServerTransport(ServerTransport transport)
  {
  }
```

Example 318: *SocketTransportFactory*

```
  public ServerTransportPolicies getServerTransportPolicies()
  {
    return serverPolicies;
  }
  private class DemoServerTransportPolicies implements ServerTransportPolicies
  {
    public void setThreadingResourcesPolicy(ServerTransportThreadingResourcesPolicy policy)
    {
    }

    public ServerTransportThreadingResourcesPolicy getThreadingResourcesPolicy()
    {
      return ServerTransportThreadingResourcesPolicy.ARTIX_DRIVEN;
    }

    public void setMessagingPortThreadingPolicy(ThreadingModel policy)
    {
    }

    public ThreadingModel getMessagingPortThreadingPolicy()
    {
      return ThreadingModel.MULTI_THREADED;
    }

    public void setRequiresConcurrentDispatchPolicy(Boolean requiresConcurrentDispatch)
    {
    }

    public Boolean getRequiresConcurrentDispatchPolicy()
    {
      return Boolean.TRUE;
    }
  }
}
```

Transport Policies

Overview

Both client and server transports have policies that are used to control how the bus manages the transport and how the transport handles messages. Client transports have only one policy. The policy controls its threading model. This policy is set in the transport factory's `getClientThreadingModel()` method.

Server transports on the other hand, have three policies that need to be set. One policy, the threading policy uses the same values as the client transport. The other policies determine who controls the threads used by the transport, if the transport is able to optimize its calls to the messaging chain, and if the transport requires all calls to be handled synchronously or asynchronously.

Transport threading models

Artix transports can use one of the three threading models listed in Table 34.

Table 34: *Transport Threading Models*

Threading Model	Behavior
`MULTI_INSTNACE`	A new instance of the transport will be created for each thread that uses this particular type of transport.
`MULTI_THREADED`	One instance of the transport is created by the bus and all threads that use this particular type of transport use the same instance. When writing transports with this threading model, you are responsible for ensuring that the code is thread safe.
`SINGLE_THREADED`	One instance of the transport is created and only one thread can access the instance.

Server transport policies

You establish the server transport's policies in the transport factory's `getServerTransportPolicies()` method. `getServerTransportPolicies()` returns an instance of the `com.iona.jbus.ServerTransportPolicies` interface. As shown in Example 318, you need to implement this interface for a custom transport.

`ServerTransportPolicies` has getter and setter methods for each of the server transport policies. You only need to provide implementaitons for the getter methods of the interface. For each policy, the value returned in the getter method is the value that the bus will use to set-up the transport. So the transport in Example 318 has the following policy settings:

- Message port threading policy is `MULTI_THREADED`.
- Threading resource policy is `ARTIX_DRIVEN`.
- Requires concurrent dispatch policy is true.

Message port threading policy

The message port threading policy determines the threading model used by the server transport. It is set in `ServerTransportPolicies.getMessagePortThreadingPolicy()`. It takes the same values as the client transport threading model. For more information see, "Transport threading models" on page 530.

Threading resource policy

The threading resource policy determines from where the threads used by the server transport are provided. It is set in `ServerTransportPolicies.getThreadingResourcePolicy()`. Server transports can either use threads provided by the bus from an Artix managed thread pool, it can directly access the bus' work queue thread, or it can manage its own thread pool.

Artix includes a static class called `com.iona.jbus.ServerTransportThreadingResourcesPolicy` that contains the values for the threading resource policy. Table 35 explains these values.

Table 35: *Threading Resource Policy Values*

Policy Value	Description
`ARTIX_DRIVEN`	Artix provides the transport with threads for processing requests. When using this setting, you may need to implement the `run()` method of the `ServerTransport` class depending on the setting of the message port threading policy.
`USES_WORKQUEUE`	Artix provides the transport with one of its work queues. The work queue will then process the incoming requests asynchronously.
`TRANSPORT_DRIVEN`	The transport is responsible for providing its own thread pool. It is also fully responsible for processing all incoming requests and ensuring that responses are returned to the client.

Requires concurrent dispatch policy

The requires concurrent dispatch policy specifies if the transport can handle concurrent requests. The setting is used by Artix to determine what optimizations can be made when processing requests. It is set using `ServerTransportPolicies.getRequiresConcurrentDsipatchPolicy()`.

Setting the requires concurrent dispatch policy to `true` informs Artix that multiple threads can call the transport's `dispatch()` method at one time. Setting it to `false` will inform Artix that the transport can process only one `dispatch()` call at a time.

Registering and Unregistering a Transport Factory

Register the transport factory

You must register the transport factory for your transport with the bus before it can be used. You register the transport factory in the `busInit()` method of the plug-in that loads your transport. The method for registering a transport factory with the bus is `bus.registerTransportFactory()`. `registerTransportFactory()` takes two arguments. The first is the namespace under which the transport will be registered. The second is an instance of the transport's transport factory.

Unregister the transport factory

When your transport is no longer needed, it should be unregistered by the transport plug-in's `busShutdown()` method. You unregister a transport using the `bus.deregisterTransportFactory()`. `deregisterTransportFactory()` takes the namespace of the transport to be unregistered as its only argument.

Example

Example 319 shows a transport plug-in that registers and unregisters a transport factory with the bus.

Example 319: *Transport Plug-in*

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

import java.net.*;
import java.io.*;

public class DemoTransportPlugIn extends BusPlugIn
{
  public DemoTransportPlugIn(Bus bus)
  {
    super(bus);
  }
```

Example 319:*Transport Plug-in*

```
  public void busInit() throws BusException
  {
    TransportFactory factory = new SocketTransportFactory();
    getBus().registerTransportFactory(
                     "http://widgetVendor.com/transport/socket",
                      factory);
  }

  public void busShutdown() throws BusException
  {
    getBus().deregisterTransportFactory(
                    "http://widgetVendor.com/transport/socket");

}
```

For more information on plug-in development see "Developing Plug-Ins" on page 469.

Developing the Client Transport

Overview

The client transport is invoked by client proxies. It is responsible for writing requests to a server and for passing the response, if one is expected, back to the proxy's binding. Requests are received from the binding, or the last request-level handler if any exists, as a stream whose contents are placed on the wire for transmission. Responses are read from the wire into a stream that is passed back up through the messaging chain.

You create a client transport by implementing the `com.iona.jbus.ClientTransport` interface. `ClientTransport` has six methods that need to be implemented. describes them.

Table 36: *ClientTransport Methods*

Method	Description
`initialize()`	Parses the Artix contract to get the initial configuration for the endpoint and initializes any resources needed by the client transport.
`connect()`	Establishes the connection between the transport and the physical hardware responsible for carrying the message.
`disconnect()`	Disables the connection and releases any system resources used by the connection.
`getOutputStream()`	Creates an output stream to which outgoing data written.
`invoke()`	Writes information out to the network and waits for a response from the server.
`invokeOneway()`	Performs similar duties to `invoke()` but it is called when the operation is defined as a oneway operation in the endpoints contract. It writes the request out to the network, but does not wait for a response.

Initializing a client transport

The `initialize()` method of the client transport is responsible for initializing any resources needed by the transport and for determining the transports initial settings. The signature for `initialize()` is shown in Example 320.

Example 320: *initialize()*

```
void initialize(String wsdlPath, QName serviceName,
                String wsdlPortName)
throws BusException;
```

It takes three parameters: `wsdlPath` is the absolute path to the Artix contract containing the transport details to be used in configuring the connection. `serviceName` is the `QName` of the service containing the definition for the endpoint. `wsdlPortName` is the name of the port defining the details of the endpoint.

The transport details of an endpoint are specified using a `port` element in an application's Artix contract and your client transport will need to parse the contract to get the information defined in this `<port>` element. The elements in which the transport details are placed should correspond to the elements defined in the previous step. You can parse the Artix contract for these elements using any XML parsing API at your disposal.

For example, the custom transport demo shipped with Artix creates a DOM for the Artix contract and parses the DOM using standard Java APIs. The demo parses the contract in following steps:

1. Find the `service` element with the service name specified by `serviceName`.
2. Find the `port` element specified by `wsdlPortName`.
3. Get the `address` element from the port.
4. Get the value for the `port` attribute.
5. Get the value for the `host` attribute.

Your transport will also need to perform steps one and two to get the `port` element defining the specifics for the endpoint. However, the rest of the parsing will be determined by the structure of the elements you defined to contain the description of an endpoint using your transport.

Example 321 shows the `initialize()` method for the custom transport demo.

Example 321: *Initialization Method for Custom Transport*

```
public void initialize(String wsdlPath, QName serviceName,
                       String wsdlPortName) throws BusException
{
1   try
    {
      DocumentBuilderFactory factory =
     DocumentBuilderFactory.newInstance();
      factory.setNamespaceAware(true);
      DocumentBuilder builder = factory.newDocumentBuilder();
      File file = new File(new URI(wsdlPath));
      Document wsdl = builder.parse(file);

2     NodeList nodes =
     wsdl.getElementsByTagNameNS("http://schemas.xmlsoap.org/wsdl/
     ", "service");

      Element serviceEl = null;

      for(int i = 0; i < nodes.getLength(); ++i)
      {
        serviceEl = (Element)nodes.item(i);
        String name = serviceEl.getAttribute("name");
        if(serviceName.getLocalPart().equals(name))
        {
          break;
        }
      }
```

Example 321: *Initialization Method for Custom Transport*

```
3     nodes =
     serviceEl.getElementsByTagNameNS("http://schemas.xmlsoap.org/
     wsdl/", "port");

     Element portEl = null;

     for(int i = 0; i < nodes.getLength(); ++i)
     {
       portEl = (Element)nodes.item(i);
       String name = portEl.getAttribute("name");
       if(wsdlPortName.equals(name))
       {
         break;
       }
     }

4    nodes =
     portEl.getElementsByTagNameNS("http://schemas.iona.com/transp
     orts/socket", "address");

     Element addressEl = (Element)nodes.item(0);

5    String port = addressEl.getAttribute("port");
     // m_portnum is defined elsewhere in this class.
     m_portnum = (new Integer(port)).intValue();

6    // m_host is defined elsewhere in this class.
     m_host = addressEl.getAttribute("host");
   }
   catch(Exception ex)
   {
     throw new BusException(ex);
   }
}
```

The code in Example 321 does the following:

1. Loads the application's contract into the DOM.
2. Finds the correct `service` element.
3. Finds the correct `port` element.
4. Finds the `address` element that defines the connection information for a port using the custom transport.
5. Sets the transport's port number to the value set in the `port` attribute.

6. Sets the transport's hostname to the value set in the `host` attribute.

Making and breaking connections in a transport

Client transport connections are made when the bus invokes the transport's `connect()` method. Its signature is shown in Example 322. `connect()` is called immediately after `initialize()` and is only called once per transport instance.

Example 322: *connect()*

```
void connect() throws BusException
```

Client transport connections are broken when the bus invokes the transport's `disconnect()` method. Its signature is shown in Example 323. `disconnect()` is called just before the bus destroys the resources used by the transport's plug-in.

Example 323: *disconnect()*

```
void disconnect() throws BusException
```

Example 324 shows code for making and breaking a socket connection.

Example 324: *Making and Breaking a Socket Connection*

```
public void connect() throws BusException
{
  try
  {
    // m_socket is defined elsewhere in this class.
    mySocket = SocketChannel.open();
    mySocket.connect(new InetSocketAddress(m_host, m_portnum));
    mySocket.finishConnect();
  }
  catch(IOException ioex)
  {
    throw new BusException(ioex);
  }
}
```

Example 324: *Making and Breaking a Socket Connection*

```
public void disconnect() throws BusException
{
  try
  {
    mySocket.close();
  }
  catch(IOException ioex)
  {
    throw new BusException(ioex);
  }
}
```

Getting an output stream

When a client proxy invokes an operation, the bus passes the request message down the messaging chain until it reaches the client transport. At this point, Artix needs a Java `OutputStream` to use for writing the request out to the wire. The client transport's `getOutputStream()` method is responsible for instantiating the output stream to which the request is written. So, when creating your transport you will need to create the appropriate type of stream for your transport. For example, the custom transport demo creates socket streams to read and write data.

`getOutputStream()`, shown in Example 325, is called imediately before the bus calls `invoke()` or `invokeOneway()`.Once `getOutputStream()` returns, the bus writes the request message into the returned output stream and then calls the proper invocation method on the transport.

Example 325: *getOutputStream()*

```
OutputStream getOutputStream(MessageContext context)
throws TransportException;
```

Example 326 shows the `getOutputStream()` implementation in custom transport demo.

Example 326: *Custom Transport Demo getOutputStream()*

```
private static final String CLIENT_TRANSPORT_CONTEXT_KEY =
  DemoClientTransport.class.getName() + ".SOCKET";

public OutputStream getOutputStream(MessageContext context)
throws TransportException
{
  try {
    Socket socket = new Socket(m_host, m_portnum);
    context.setProperty(CLIENT_TRANSPORT_CONTEXT_KEY, socket);
    return socket.getOutputStream();
  } catch (IOException ioex) {
    throw new TransportException(ioex);
  }
}
```

Invoking an operation

After writing the request, the bus calls either the client transport's `invoke()` method or the client transport's `invokeOneway()` method depending upon how the operation is defined in the application's contract.

The bus calls `invoke()` when the operation definition in the application's contract has both an input message and an output message. If the operation is defined as a oneway operation, meaning that it only has an input message, then the bus calls `invokeOneway()`.

Both operations receive the `OutputStream` to which the bus wrote the request and the `MessageContext` object associated with the invocation. Depending on the type of output stream used, invoke() and invokeOneway() may need to push the request out to the wire. For example, a transport the uses `ByteArrayOutputStream` output streams will need to push the data to the wire. However, if the transport uses a socket output stream, like the custom transport demo, the data is pushed to the wire as soon as it is written into the output stream.

Note: For information on accessing information in a message context, see "Using Message Contexts" on page 267.

The difference between the operations is that `invoke()` waits for a response to be returned and passes the response back the bus as a Java `InputBuffer`. `invokeOneway()` simply returns after pushing the message to the wire.

The signatures for `invoke()` and `invokeOneway()` are shown in Example 327.

Example 327: *Invoking Operations From the Transport*

```
InputStream invoke(OutputStream request, MessageContext context)
throws TransportException
void invokeOneway(OutputStream request, MessageContext context)
throws TransportException
```

Example 328 shows he implementation of `invoke()` used in the custom transport demo. The code gets the socket created for the invocation in `getOutputStream()`. It then gets the response from the socket as an `InputStream`.

Example 328: *invoke() for a Socket Transport*

```
public InputStream invoke(OutputStream request,
                          MessageContext context)
throws TransportException
{
  try {
    final Socket socket =
   (Socket)context.getProperty(CLIENT_TRANSPORT_CONTEXT_KEY);
    socket.shutdownOutput();

    //close the socket when done
    return new FilterInputStream(socket.getInputStream()) {
      public void close() throws IOException {
        super.close();
        socket.close();
      }
    };
  } catch (IOException ioex) {
    throw new TransportException(ioex);
  }
}
```

Developing the Server Transport

Overview

The server transport is responsible for reading requests from the wire, passing it to the server binding, and then writing the replies back to the wire for delivery. Requests are read from the wire using input streams that are passed on to any request-level handlers and then to the binding. Replies are returned to the transport as an output stream that is then placed back on the wire.

You create a server transport by implementing the `com.iona.jbus.ServerTransport` interface. `ServerTransport` has six methods as shown in Table 37.

Table 37: *ServerTransport Methods*

Method	Description
`activate()`	Parses the Artix contract to get the initial configuration for the endpoint and initializes any resources needed by the server transport. If the transport's message port threading policy is `MULTI_INSTANCE` and the transport's threading resource policy is `ARTIX_DRIVEN`, `activate()` is also responsible for request processing.
`run()`	Reads requests off of the wire and dispatches them to the transport callback object. The callback object then passed the message up the messaging chain.
`getOutputStream()`	Creates the output stream to which the bus writes responses.
`postDispatch()`	Called by the transport callback object after it writes the response to the output stream. Depending on the type of output stream used, `postDispacth()` may have to push the response to the wire. `postDispatch()` can also be used to clean up any resources used in processing the request.

Table 37: *ServerTransport Methods*

Method	Description
`deactivate()`	Stops the transport listener and allows any requests that are already in process to complete.
`shutdown()`	Disables the connection and releases any system resources used by the connection.

Depending on the server transport policies set for the transport, you do not need to implement all of the methods. At a minimum, you will need to provide implementations for `activate()`, `getOutputStream()`, `deactivate()`, and `shutdown()`.

In this section

This section discusses the following topics:

Activating a Server Transport

Overview

The `activate()` method of the server transport is responsible for initializing any resources needed by the transport and for determining the transports initial settings. Depending on the threading policies set on the transport, `activate()` may also have other responsibilities such as request processing.

activate()

The signature for `activate()` is shown in Example 320.

Example 329: *activate()*

```
void activate(String wsdlPath, QName service, String port,
              TransportCallback callback, WorkQueue queue)
throws TransportException
```

`activate()` takes five parameters: `wsdlPath` is the absolute path to the Artix contract containing the transport details to be used in configuring the connection. `serviceName` is the `QName` of the service containing the definition for the endpoint. `port` is the name of the port defining the details of the endpoint. `callback` is a reference to a bus managed callback object that passes the request up the message chain and returns the output stream containing the reply. `queue` is the Artix `WorkQueue` that will be used by the transport to process requests if the threading resource policy is set to `USES_WORKQUEUE`.

Note: You do not need to implement the callback object because it is implemented and managed by the bus. However, your transport does need to maintain a handle to the callback object to pass requests up the message chain.

Contract parsing

The transport details of an endpoint are specified using a `port` element in an application's Artix contract and your client transport will need to parse the contract to get the information defined in this `port` element. The elements in which the transport details are placed should correspond to the elements defined in the previous step. You can parse the Artix contract for these elements using any XML parsing API at your disposal.

For example, the custom transport demo shipped with Artix creates a DOM for the Artix contract and parses the DOM using standard Java APIs. The demo parses the contract in following steps:

1. Find the `service` element with the service name specified by `serviceName`.
2. Find the `port` element specified by `wsdlPortName`.
3. Get the `address` element from the port.
4. Get the value for the `port` attribute.
5. Get the value for the `host` attribute.

Your transport will also need to perform steps one and two to get the `port` element defining the specifics for the endpoint. However, the rest of the parsing will be determined by the structure of the elements you defined to contain the description of an endpoint using your transport.

Threading policies and activate()

The threading policies set on the server transport will determine, to some extent, how you code `activate()`. In all cases, `activate()` will need to parse the contract and set-up the transport's resources. However, the threading policy settings determine what `activate()` needs to do after the transport resources are set-up.

Table 38 shows what `activate()` needs to do for all combinations of message port threading policy settings and threading resource policy settings.

Table 38: *activate() Responsibilities by Threading Policies*

Message Port Thread Policy	Threading Resource Policy	activate() Responsibilities
`MULTI_THREADED`	`USES_WORKQUEUE`	`activate()` spawns a new thread to host the `WorkQueue` provided by the `queue` parameter. The new thread processes requests.
`MULTI_INSTANCE`	`USES_WORKQUEUE`	
`SINGLE_THREADED`	`USES_WORKQUEUE`	
`MULTI_THREADED`	`ARTIX_DRIVEN`	`activate()` can exit once the transport's resources are set-up.
`MULTI_INSTANCE`	`ARTIX_DRIVEN`	`activate()` must block and process requests from the wire.

Table 38: *activate() Responsibilities by Threading Policies*

Message Port Thread Policy	Threading Resource Policy	activate() Responsibilities
SINGLE_THREADED	ARTIX_DRIVEN	activate() can exit once the transport's resources are set-up.
MULTI_THREADED	TRANSPORT_DRIVEN	activate() creates the threads used by the transport to process requests and hands control off to them.
MULTI_INSTANCE	TRANSPORT_DRIVEN	
SINGLE_THREADED	TRANSPORT_DRIVEN	

Notifying the bus

Once the server transport is activated, the transport needs to inform the bus that the transport is going to begin dispatching messages. The transport callback object's `transportActivated()` method notifies the bus that the transport is active and ready to begin dispatching messages up the message chain. `transportActivated()` must be called before you begin dispatching messages.

Example

Example 330 shows the `activate()` method for the custom server transport demo. The transport used in the custom transport demo uses the `MUTLI_THREADED` message port threading policy and the `ARTIX_DRIVEN` threading resource policy. Therefore, it does not use the `WorkQueue` passed into it and does not block.

Example 330:*Activation Method for Custom Server Transport*

```
// Java
import com.iona.jbus*;
...

public class SocketServerTransport implements ServerTransport
{
private TransportCallback theCallback;
private ServerSocket serverSocket;
...

public void activate(String wsdlPath, QName serviceName,
                     String wsdlPortName,
                     TransportCallback callback, WorkQueue queue)
throws TransportException
{
```

Example 330:*Activation Method for Custom Server Transport*

```
1  theCallback = callback;

2  try
   {
     DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
     factory.setNamespaceAware(true);
     DocumentBuilder builder = factory.newDocumentBuilder();
     File file = new File(new URI(wsdlPath));
     Document wsdl = builder.parse(file);

3    NodeList nodes =
    wsdl.getElementsByTagNameNS("http://schemas.xmlsoap.org/wsdl/
    ", "service");

     Element serviceEl = null;

     for(int i = 0; i < nodes.getLength(); ++i)
     {
       serviceEl = (Element)nodes.item(i);
       String name = serviceEl.getAttribute("name");
       if(serviceName.getLocalPart().equals(name))
       {
         break;
       }
     }

4    nodes =
    serviceEl.getElementsByTagNameNS("http://schemas.xmlsoap.org/
    wsdl/", "port");

     Element portEl = null;

     for(int i = 0; i < nodes.getLength(); ++i)
     {
       portEl = (Element)nodes.item(i);
       String name = portEl.getAttribute("name");
       if(wsdlPortName.equals(name))
       {
         break;
       }
     }
```

Example 330: *Activation Method for Custom Server Transport*

```
5     nodes =
     portEl.getElementsByTagNameNS("http://schemas.iona.com/transp
     orts/socket", "address");

     Element addressEl = (Element)nodes.item(0);

6     String port = addressEl.getAttribute("port");
     int portnum = (new Integer(port)).intValue();

7     String host = addressEl.getAttribute("host");

8     serverSocket = new ServerSocket(portnum, 0,
                                      InetAddress.getByName(host));

9     theCallback.transportActivated();
   }
   catch(Exception ex)
   {
     throw new TransportException(ex);
   }
...
}
```

The code in Example 321 does the following:

1. Saves a handle to the transport callback in a private data member.
2. Loads the application's contract into the DOM.
3. Finds the correct `service` element.
4. Finds the correct `port` element.
5. Finds the `address` element that defines the connection information for a port using the custom transport.
6. Sets the transport's port number to the value set in the `port` attribute.
7. Sets the transport's hostname to the value set in the `host` attribute.
8. Creates a `ServerSocket` to connect to the endpoint.
9. Notifies the bus that the transport is active and ready to dispatch messages.

Processing Requests

Overview

Server transport process requests by reading the data off of the wire, dispatching the request to the transport callback object in an input stream, and then writing the response to the wire. Which method is responsible for reading the request from the wire and dispatching the request to the transport callback object depends on the transport's policy settings. For example, in a mulit-instance transport with a thread resource policy of `ARTIX_DRIVEN`, reading the request and dispatching the request to the transport callback would be handled in `activate()`. However, in a transport with a thread resource policy of `USES_WORKQUEUE`, the message reading is done in a `WorkItem` object.

The method responsible for writing the response to the wire depends on the type of output stream used to write the response. If you use an output stream that automatically writes the message to the wire, such as a socket output stream or a file output stream, the request is put on the wire when the transport callback puts the message into the output stream. However, if your transport uses an output stream type that does not write to the wire, such as a `ByteArrayOutputStream`, `postDispatch()` will need to push the response to the wire. See "Writing the response" on page 554.

Dispatching messages to the messaging chain

Server transports use a callback mechanism to pass messages to the messaging chain. The `TransportCallback` object provided to `activate()` is used to dispatch requests to the messaging chain and return the responses. The `TransportCallback` object has one method `dispatch()` that takes an input stream containing a request message and the active `MessageContext` object as input parameters. The signature for `dispatch()` is shown in Example 331.

Example 331: *TransportCalback.dispatch()*

```
void dispatch(InputStream request, MessageContext ctx);
```

When the message chain returns the response to the transport callback object, the transport callback object calls `getOutputStream()` on the server transport to get an output stream. The transport callback object writes the response into the returned output stream and then calls `postDispatch()` on the server transport. See "Writing the response" on page 554.

Reading requests with a USES_WORKQUEUE threading resource policy

When a transport's threading resource policy is set to `USES_WORKQUEUE`, you implement a thread to read requests off of the wire and place them on the `WorkQueue`. The requests are dispatched to the messaging chain by a `WorkItem` object that you implement.

The first step is to extend the `Thread` class for your transport. In the thread's `run()` method, three things need to happen.

1. Requests are read into an input stream.
2. The stream is packed into a `WorkItem` object.
3. The `WorkItem` is placed onto the work queue using the work queue's `enqueue()` method.

Example 332 shows a thread for a server transport with a threading resource policy of `USES_WORKQUEUE`.

Example 332: *Server Transport Thread*

```
class demoListenerThread extends Thread
{
  private final WorkQueue theQueue;
  private final Socket theSocket;
  private final TransportCallback theCallback;

  public listenerThread(WorkQueue workQueue,
                        ServerSocket serverSocket,
                        TransportCallback callback)
  {
    theQueue = workQueue;
    theSocket = serverSocket.accept();
    theCallback = callback;
  }

  public void run()
  {
    while (true)
    {
      InputStream request = theSocket.getInoutStream();
      WorkItem item = new demoWorkItem(request, theCallback);
      theQueue.enqueue(item, -1);
    }
  }
}
```

The second thing you need to do is implement the `com.iona.jbus.WorkItem` interface for your transport. `WorkItem` has two methods: `execute()` and `destroy()`.

`execute()` is called when the work queue processes this work item. In `execute()`, your work item needs to dispatch the request message to the messaging chain using the transport callback's `dispatch()` method.

`destroy()` is called by the work queue when the work item is finished being processed. It is is responsible for cleaning up any resources used by the work item.

Example 333 shows a work item for a server transport.

Example 333: *Transport Work Item*

```
import com.iona.jbus.BusException;
import com.iona.jbus.WorkItem;

public class demoWorkItem implements WorkItem
{
  private final TransportCallback theCallback;
  private final ByteBuffer theMessage;

  public demoWorkItem(InputStream message,
                      TransportCallback callback)
  {
    theMessage = message;
    theCallback = callback;
  }

  public void execute() throws BusException
  {
   MessageContext context = theCallback.getCurrentContext();
   theCallback.dispatch(requestBuf, context);
  }

  public void destroy() throws BusException
  {
  }
}
```

Reading requests with a ARTIX_DRIVEN threading resource policy

When a transport's threading resource policy is set to `ARTIX_DRIVEN` and its message port threading policy is set to `MULTI_THREADED`, `run()` is responsible for pulling requests off of the wire and dispatching them to the

messaging chain. `run()` is called once per thread that uses the transport and must loop for as long as the connection is open. Inside the loop, `run()` reads requests off of the wire and passes the requests up the messaging chain using the transport callback's `dispatch()` method.

When a transport's threading resource policy is set to `ARTIX_DRIVEN` and its message port threading policy is set to `MULTI_INSTANCE`, `activate()` is responsible for pulling requests off of the wire and dispatching them to the transport callback method. In this case, `activate()` must block by looping as long as the connection is open. Inside the loop, `activate()` reads requests off the wire and dispatching them to the messaging chain.

Example 334 shows the code for implementing `run()` for a multi-threaded transport.

Example 334: *run() for a Custom Server Transport*

```
// Java
import iona.com.jbus.*;

public class SocketServerTransport implements ServerTransport
{
...

  public void run() throws TransportException
  {
    try
    {
      ++connectionCount;

1     while (!serverSocket.isClosed())
      {
        Socket socket;

2       synchronized(serverSocket)
        {
3         if (!serverSocket.isClosed())
          {
            socket = serverSocket.accept();
          } else
          {
            break;
          }
        }
4       MessageContext dispatchContext =
   theCallback.getCurrentContext();
```

Example 334: *run() for a Custom Server Transport*

```
5         dispatchContext.setProperty(SERVER_TRANSPORT_CONTEXT_KEY,
                                      socket);

6           theCallback.dispatch(socket.getInputStream(),
                                 dispatchContext);
        }
      } catch (Exception ex)
      {
        throw new TransportException(ex);
      }
    }
}
```

The code in Example 334 does the following:

1. Loop for as long as the socket opened in `activate()` remain open.
2. Synchronizes access to the socket to ensure thread safety.
3. Blocks until a socket channel is accepted.
4. Gets the message context.
5. Stores the socket in the message context for later use.
6. Dispatches the request to the transport callback object.

Reading requests with a TRANSPORT_DRIVEN threading resource policy

When the threading resource policy is set to `TRANSPORT_DRIVEN`, your transport is responsible for implementing its own threads for processing messages. The implementation details would be similar to implementing a transport with the `USES_WORKQUEUE` threading resource policy. In your thread's `run()`, you would pull messages off of the wire and dispatch them to the messaging chain using the transport callback object. Where the response were written to the wire would depend on the type of output streams used and how your transport pushes data to the wire.

Writing the response

When the message chain returns a response to the transport callback object, the transport callback object does the following:

1. Invokes `getOutputStream()` on the server transport to get an appropriate output steam for writing the response.
2. Writes the response into the returned output stream.

3. Invokes `postDispatch()` on the server transport to allow for any post processing that need to be done.
4. Closes the output stream.

You are responsible for providing implementations of `getOutputStream()` and `postDispatch()` for your server transport.

`getOutputStream()`, as shown in Example 335, takes a message context as a parameter and returns a Java `OutputStream` into which the transport callback object will write the response.

Example 335: *ServerTransport.getOutputStream()*

```
public OutputStream getOutputStream(MessageContext ctx)
throws TransportException;
```

Example 336 shows the implementation of `getOutputStream()` used in the custom transport demo. It creates a socket output stream using a socket stored in the request's message context. The resulting output stream provides a direct connection to the client who made the request.

Example 336: *Socket Transport Server Side getOutputStream()*

```
public OutputStream getOutputStream(MessageContext ctx)
throws TransportException
{
  try
  {
    Socket socket =
   (Socket)ctx.getProperty(SERVER_TRANSPORT_CONTEXT_KEY);
    return socket.getOutputStream();
  } catch (Exception ex)
  {
    throw new TransportException(ex);
  }
}
```

`postDispatch()` is called by the transport callback object after the response is written to the output stream. It is used to do any post-processing and clean-up required after a request is fully processed. As shown in Example 337, `postDispatch()` takes the `OutputStream` containing the response and the request's message context.

Example 337: *postDispatch()*

```
public void postDispatch(OutputStream request,
                         MessageContext ctx)
throws TransportException;
```

shows the implementation of `postDispatch()` used in the custom transport demo. Because this transport uses socket streams, `postDispatch()` does not need to do anything to with the output stream. The response was delivered when the transport callback object wrote it to the output stream. However, if your transport uses some other mechanism for pushing the response to the wire, `postDispatch()` would be the method to place that logic.

Example 338: *Custom Transport postDispatch()*

```
public void postDispatch(OutputStream request,
                         MessageContext ctx)
throws TransportException
{
  try
  {
    Socket socket =
   (Socket)ctx.getProperty(SERVER_TRANSPORT_CONTEXT_KEY);
    socket.close();
  } catch (Exception ex)
  {
    throw new TransportException(ex);
  }
}
```

Using message contexts

If your transport uses a header block to pass transport information, like the header used by JMS, that the application code may be interested in, you can pass this information up the messaging chain using the Artix message context mechanism.

To get access to the application's message context, you use the `getCurrentContext()` method of the transport callback object. `getCurrentContext()` returns a JAX-RPC `MessageContext` object. To pass custom header information back to the application level, you will need to cast the JAX-RPC message context to an `IonaMessageContext` object and set the appropriate context properties. The transport callback will automatically pass the context information up the messaging chain where the handlers and application level code can access it.

For more information on using contexts see "Using Message Contexts" on page 267.

Shutting Down a Server Transport

Overview

When the bus shuts a servant down it calls `shutdown()` on the transports used by that servant. `shutdown()` is responsible for closing any open connections used by the transport and cleaning up the resources used by the transport.

Shutting down a transport using a TRANSPORT_DRIVEN threading resource policy

When your transport uses the `TRANSPORT_DRIVEN` threading resource policy, Artix does not automatically clean up the transport's threads. Your `shutdown()` implementation must clean-up all of the threads spawned by the transport.

Notifying the bus

When the transport has finished cleaning up its resources and is ready to be fully shutdown, it need to notify the bus that it can no longer send or receive messages. The transport callback's `transportShutdownComplete()` method notifies the bus when the transport is done shutting itself down and cannot accept any more messages. Typically this is the last thing your server will do before `shutdown()` exits.

Example

Example 339 shows the code used to disconnect a socket server transport. The code simply loops through all of the open sockets and closes them. Once the sockets are closed the loop in `connect()` is broken and it will exit.

Example 339:*Disconnecting a Custom Server Transport*

```
// Java
import iona.com.jbus.*;

public class SocketServerTransport implements ServerTransport
{
...
```

Example 339:*Disconnecting a Custom Server Transport*

```
  public void disconnect() throws Exception
  {
    if(--connectionCount <=0)
    {
      m_SSChannel.close();
    }

    m_callback.transportShutdownComplete();
  }
}
```

Using your Custom Transport

Overview

To use a custom transport you need to add the appropriate entries in you application's contract and add some configuration to your Artix configuration file. The entries in the application's contract inform the bus that your application uses the transport and describes how the endpoint is to be established. The configuration information tells Artix how to load the plug-in that implements the transport.

Adding the transport to an Artix contract

To make an application use your custom transport, you must create an endpoint that is defined as using the custom transport in the application's contract. You add an endpoint description to a contract in two steps:

1. Add an XML namespace declaration to the `definition` element of the contract so that the contract can include elements defined by the schema defining your transport.
2. Add a `service` element and `port` element to describe an endpoint that uses your transport to the contract.

Example 340 shows a fragment from a contract that uses the custom socket transport defined in this chapter. Notice that the namespace declaration for the socket transport, `xmlns:sock="http://widgetVendor.com/transport/socket"`, uses the target namespace from the schema definition of defining the WSDL extensions for describing a the transport.

Example 340:*Contract using a Custom Transport*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetSocketVendor"
 targetNamespace="http://schemas.iona.com/widgetVendor"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.iona.com/widgetVendor"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:sock="http://widgetVendor.com/transport/socket"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
...
```

Example 340:*Contract using a Custom Transport*

```
  <service name="widgetService">
    <port binding="tns:widgetSOAPBinding" name="widgetPort">
      <sock:address host="localhost" port="8080"/>
    </port>
  </service>
</definitions>
```

For more information on defining endpoints in an Artix contract see Designing Artix Solutions.

Configuring Artix to load the transport

To use a custom transport plug-in, you must make three modifications to the application's configuration:

1. Add the Java plug-in to your application's `orb_plugins` list.
2. Specify the namespace for the transport plug-in in the global scope of the Artix configuration file.
3. Specify the plug-in factory for the plug-in that implements the plug-in.

Specifying the namespace for a transport plug-in

The bus identifies which transport plug-ins to load based on the endpoints defined in an application's contract. To do this the bus looks through its configuration for a namespace match and then loads the specified plug-in. The namespaces are specified using variables pre-fixed with `namespace` and have the syntax shown in Example 341.

Example 341:*Specifying a Transport Namespace*

```
namespace:xml_namespace:plugin="plugin_name";
```

`xml_namespace` is the target namespace in the XMLSchema used to define your transport's attributes. *`plugin_name`* is the name by which the plug-in is configured in the Artix configuration file. For example to specify the namespace for the socket transport implemented in this chapter you would use a configuration entry similar to Example 342.

Example 342:*Socket Transport Namespace Specification*

```
namespace:http://widgetVendor.com/transport/socket:plugin="sock"
   ;
plugin:sock:classname="SocketPluginFactory";
```

For more information on configuring Artix plug-ins see "Configuring Artix Plug-Ins" on page 563.

CHAPTER 22

Configuring Artix Plug-Ins

Artix plug-ins can use the Artix runtime configuration file to receive configuration information.

In this chapter

This chapter discusses the following topics:

Understanding Artix Configuration

Overview

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

In this section

This section discusses the following:

Configuration domains

An Artix *configuration domain* is a collection of configuration information in an Artix runtime environment. This information consists of configuration variables and their values. A default Artix configuration is provided when Artix is installed. The default configuration file is located in:

Windows `%IT_PRODUCT_DIR%\artix\`*`artix_version`*`\etc\domains\artix.cfg`

UNIX `$IT_PRODUCT_DIR/artix/`*`artix_version`*`/etc/domains/artix.cfg`

You can also manually create new Artix configuration domains to compartmentalize your applications. These new configuration domains can import information from other configuration domains using a `#include` statement in your configuration. This provides a convenient way of compartmentalizing your application specific configuration from the global Artix configuration information contained in the default domain.

Configuration scopes

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

Applications read their configuration information from a given scope based on the ORB name passed into the application's `bus.init()` call. Application-specific configuration variables either override default values assigned to common configuration variables, or establish new configuration variables.

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

Example 343 shows the nested configuration scope `demo`. In each nested scope, `orb_plugins` is redefined so that an application starting up in one scope will load a different set of plug-ins from one starting in another scope. In addition, each scope sets application-specific configuration variables.

Example 343: *Demo Configuration Scope*

```
demo
{
 fml_plugin
 {
     orb_plugins = ["local_log_stream"];
 };
 telco
 {
    orb_plugins = ["xml_log_stream", "router"];
    plugins:tunnel:iiop:port = "55002";
    poa:MyTunnel:direct_persistent = "true";
    poa:MyTunnel:well_known_address = "plugins:tunnel";

    server
      {
        orb_plugins = ["local_log_stream", "iiop_profile",
                       "giop", "iiop", "ots"];
        plugins:tunnel:poa_name = "MyTunnel";
     };
 };
}
```

Specifying configuration scopes

To make an Artix process run under a particular configuration scope, you specify that scope using the `-ORBname` parameter. Configuration scope names are specified using the format *scope*`.`*subscope*.

For example, the scope for the telco server demo shown in Example 343 is specified as `demo.telco.server`. During process initialization, Artix searches for a configuration scope with the same name as the `-ORBname` parameter.

There are two ways of supplying the `-ORBname` parameter to an Artix process:

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

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

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

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

```
bus.init (argc, argv, "demo.tibrv");
```

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

Configuration namespaces

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

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

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

```
plugins:routing:wsdl_url
```

Configuration variables

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

Variables can also be reset several times within successive layers of a configuration scope. Configuration variables set in narrower configuration scopes override variable settings in wider scopes. For example, a company.operations.orb_plugins variable would override a

company.orb_plugins variable. Plug-ins specified at the company scope would apply to all processes in that scope, except those processes that belong specifically to the company.operations scope and its child scopes.

Configuration data types

Each configuration variable has an associated data type that determines the variable's value.

Data types can be categorized into two types:

- Primitive types
- Constructed types

Primitive types

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

Constructed types

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

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

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

Adding Custom Configuration for a Plug-in

Overview

Artix provides an API that allows you to access the Artix configuration mechanism from with in Java plug-ins. This API makes it easy to place any configuration information required by a custom plug-in into the standard Artix configuration file.

Variable scoping

The configuration APIs search for configuration variables using fully qualified variable names similar to the ones used in the common configuration elements. This means that your custom variables are subject to the same scoping rules as common configuration elements. So, variables in local scopes override variables set in more global scopes.

Variable naming

For consistency, it is recommended that you make your configuration variable names consistent with the naming scheme applied to standard Artix configuration elements. So, the variables for your plug-ins would also use the syntax shown in Example 344.

Example 344: *Plug-in Variable Syntax*

```
plugins:plugin_name:var_name=value;
```

plugin_name is the name used to refer to the plug-in throughout the configuration file. *var_name* is the name of the configuration variable and *value* is the value of the variable.

Supported variable types

The Artix configuration APIs allow you to use either string configuration variables or list configuration variables. Example 345 shows a variable with a string value.

Example 345: *String Value*

```
plugins:junk:junkyard="\etc\junkyard";
```

Example 346 shows a variable with a list value.

Example 346: *List Value*

```
plugins:junk:filters=["spam", "adult", "blacklist"];
```

Getting the configuration

The bus provides access to the configuration using `getConfiguration()`. `getConfiguration()` returns a `Configuraiton` object that provides access to the application's configuration.

Example 347 shows code for getting the configuration in a plug-in.

Example 347: *Getting Access to Configuration Details*

```
//Java
import com.iona.jbus.*;

public void busInit() throws BusException
{
  Bus bus = getBus();

  Configuration config=bus.getConfiguration();

  ...
}
```

The code in Example 347 does the following:

1. Gets a reference to the plug-ins bus.
2. Gets the bus' configuration information.

Reading string values

To read a configuration variable with a string value you use the `Configuration` object's `getString()` method. The signature for `getString()` is shown in Example 348. If it finds the specified variable, it returns the value as a string. If it does not find the variable, it returns a null string.

Example 348: *getString()*

```
String getString(String name);
```

Example 349 shows the code for reading the variable `plugins.junk.junkyard`.

Example 349:*Reading a String Value*

```
// Java
String junkyard = config.getString("plugins:junk:junkyard");
```

Reading list values

To read a configuration variable with a list value you use the `Configuration` object's `getList()` method. The signature for `getList()` is shown in Example 348. If it finds the specified variable, it returns the entries in the list as an array of strings. If it does not find the variable, it returns a null array.

Example 350:*getString()*

```
String[] getlist(String name);
```

Example 349 shows the code for reading the variable `plugins.junk.filters` and printing out the values.

Example 351:*Reading a String Value*

```
// Java
String[] filterList = config.getList("plugins:junk:filters");

for (int i = 0; 1 < filterList.length ; ++)
{
  System.out("Filter: "+filterList[i]);
}
```

CHAPTER 23

Using Artix Classloader Environments

Artix Classloader Environments provide an easily configurable mechanism for overcoming some of the shortcomings in Java's default class loading scheme. In particular, they give you finer control over which classes are loaded by each classloader in an application's classloader chain.

In this chapter

This chapter discusses the following topics:

Class Loading: An Overview

Introduction

One of Java's most important features is that compiled Java applications are platform independent. Unlike, C++ applications, for instance, a Java application can be built on a Windows system and run without modification on a UNIX system.

Part of the mechanism used to allow this platform independence is the way the Java Virtual Machine, or JVM, loads the binary data that makes up a Java application. Java binary code is stored, at its most atomic state, as a class file that stores the binary code for a Java `Class` object. When the JVM needs to create an instance of a `Class` object it loads the class' binary representation using a classloader. The classloader reads in the binary data, transforms the data into usable machine code, and creates a generic `java.lang.Class` object for the class.

To enhance the performance of the JVM, classloaders only load a class the first time it is needed and then cache the data in case it is needed again. Classloaders are also split into a hierachical structure to provide a level of security for the JVM. This hierarchical structure prevents classloaders in the application space from loading corrupt versions of core Java classes.

When are classes loaded?

Any of the following events can trigger a class to be loaded:

- The creation of a new instance of a class.
- The dependency of one class on another class. For example, if class `Foo` has a member of class `Bar`, then `Bar` will need to be loaded along with `Foo`.
- An explicit call to a classloader's `loadClass()` method.

Classloader chaining

Classloaders link together to form a chain where each classloader holds a link to the classloader that created it. When a classloader attempts to load a class, it first checks its local cache. If the class is not in the local cache, the classloader then checks with its parent classloader to find the class. Finally, if the class has not been loaded by any of the existing classloaders, the classloader loads the class from an external source.

So, if your application has three classloaders, A, B, and C as shown in Figure 28, classloader C will always check with classloaders A and B before loading a class from an external source. For example, if class c3 has a dependency on class a1, it will not need to be loaded because it is supplied by classloader A.

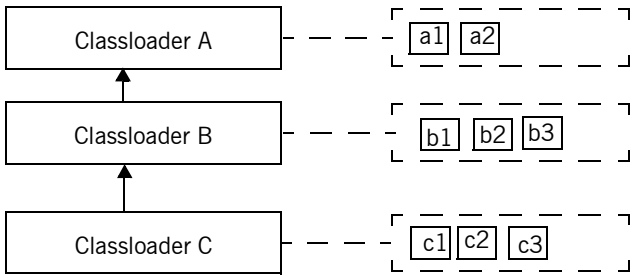

Figure 28: *Classloader Chain*

Default classloader hierachy

The JVM provides a default classloader hierachy to supply a minimal guarantee that the JVM's core classes do not get corrupted or overwritten by application specific class implementations. The JVM's classloader hierachy consists of three levels as shown in Figure 29.

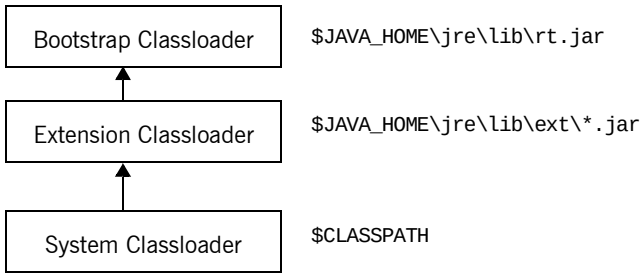

Figure 29: *Default Classloader Hierarchy*

The bootstrap classloader is responsible for loading the core Java classes such as `java.lang.Object`. The extension classloader then loads any runtime extension classes such as the ones that provide localization support. Finally, the system classloader loads the remainder of the classes needed by an application.

Limitations of classloaders

While the design of the class loading system is effective in ensuring that the core Java classes are not hijacked and isolating user defined classes based on where they are loaded, it does not address two key issues. These are:

- Using multiple versions of the same library in a single application.
- Classes becoming inaccessible.

In large applications where some of the core functionality is provided by vendor supplied libraries, you may run into a situation where multiple versions of a core library, such as Xerces or log4j, are desired. For example, the vendor supplied libraries may use Xerces 1.0 while your application code uses Xerces 2.0. In this instance, the first version of the library loaded will be the version used.

It is also possible for classes to become inaccessible because it is possible for a class may have dependencies on classes that are only available to a classloader further down the classloader chain. Because the classloader mechanism only checks up the chain, the dependencies cannot be resolved.

Artix's Classloader Hierarchy

Overview

You can configure Artix to add two additional layers to the default classloader hierachy used by the JVM when the bus or any Artix plug-in is loaded. The first is a firewall classloader that can be configured to block access to classes loaded by classloaders higher up the chain. The second is a classloader that can be configured to load all of the classes needed by the bus or the plug-in from a specific set of resources, including URLs. This is shown in Figure 30.

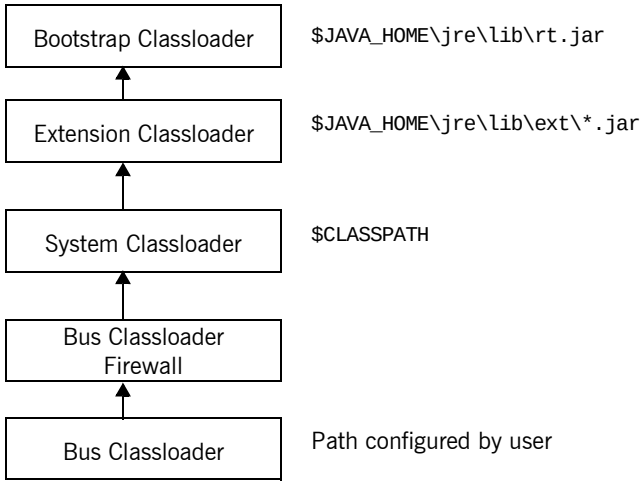

Figure 30: *Artix Bus Classloader Chain*

Why use the added classloaders?

Adding these two classloaders solves both of the problems of Java's classloader system. It solves the problem of using multiple versions of a library by blocking the bus', or the plug-in's, classloader from classes loaded by other classloaders and directing the bus', or the plug-in's, classloader to load only the version of the classes in its path. It solves the problem of

inaccessible classes in much the same way. Because the bus, or the plug-in, has a dedicated classloader, all of the classes needed by it are accessible.

In addition, the Artix classloader environment's dedicated classloader removes an application's dependency in listing all of the required classes in the CLASSPATH. You can specify where the classes to be loaded by the Artix classloader are located. The location of the resources used by the dedicated classloader can be specified using absolute paths or valid URLs. Thus you can load classes over the web or from a central repository if needed.

Where do plug-ins fit into the hierarchy?

If a plug-in is configured to use the optional Artix classloaders, the parent classloader of the plug-in's firewall classloader will be the classloader that loaded the bus as shown in Figure 31. If the bus is loaded by the system classloader, then the plug-in's firewall classloader will block classes from the system classloader and above. If the bus is configured to use the Artix

classloading environment, the bus' classloader becomes the parent classloader for the plug-in. In this instance, the plug-in will only have access to the classes that are allowed through the bus' classloader firewall.

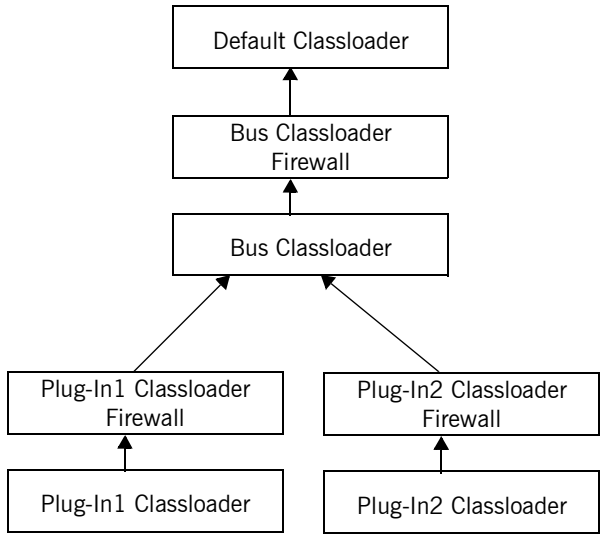

Figure 31: *Artix Plug-In Classloader Chain*

If the bus blocks a system class from the plug-ins, it create problems for the plug-ins. Therefore you must be careful when creating the rules for what is allowed through the bus' classloader firewall. Optionally, you can also use the plug-in's classloader to load the needed classes from the system. However, these loaded classes will not inherit from the class instances loaded by other plug-ins or components that are loaded by the system classloader.

Classloader chaining

If you are using multiple plug-ins that are configured to use the Artix classloader environment, or the bus itself is using the Artix classloader environment, you can specify the order in which the classloaders are placed into the classloader hieracrchy. The bus' classloader will always be the

parent of the first plug-in loaded, but the order in which the plug-in's classloaders are placed into the hierarchy can be specified in the classloader configuration files.

By default, all of the plug-in classloaders are children of the classloader that loaded the Artix bus. However, inside the each plug-in's classloader configuration you can specify which classloader will be the current classloader's parent. This can be useful if you have a number of plug-ins that share common set or restrictions or that need a particular chain of inheritance to remain intact.

Using Artix's Classloader Environment

Overview

The Artix classloader environment provides a powerful mechanism for controlling what classes are used by the Artix bus and the plug-ins that make up your applications. However, it is easy to configure. You simply add the appropriate configuration information the Artix configuration file to tell your code to use the Artix classloader environment. Then you configure the classloader firewall and resource locations in a CE file that is written in XML.

Creating the CE file

The Artix classloader environment is configured using CE files. Each plug-in that uses the Artix classloader environment will have a CE file that defines the parent of its classloader in the classloader hierarchy, the filters used by its classloader firewall, and where the its classloader looks for resources.

CE files are written in XML and use a small number of elements to define the environments behavior. Each CE file has four parts. The first part is common to all CE files and should appear in all CE files you create. It defines the encoding style used, the type of XML document being specified, and a namespace shortcut. The entries for this section are shown in Example 352.

Example 352: *CE File Preamble*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ce:classloader-environment PUBLIC "-//IONA//DTD IONA Classloading Environment 2.0//EN"
 "http://www.iona.com/dtds/classloader-environment_2_0.dtd">
<ce:classloader-environment xmlns:ce="http://www.iona.com/ns/classloader-environment"
 loglevel="info">
```

The second section is contained in the `ce:environment` element of the document. This element is the only child of the top-level `ce:classloader-environment` element. This section specifies the classloader environment's name using the `name` attribute of `ce:environment`

as shown in Example 353. In addition, you can use the optional parent attribute to define the classloader's parent as discussed in "Chaining classloaders" on page 582.

Example 353: *Naming a Classloader Environment*

```
<ce:classloader-environment>
  <ce:environment name="sifter_ce">
...
  </ce:environment>
</ce:classloader-environment>
```

The third section of the CE file defines the filters used by the classloader firewall. It consists of both positive and negative filter definitions defined inside of the `ce:firewall` element. `ce:firewall` is the first child of `ce:environment` and has one or more `ce:filter` child elements. Defining firewall filters is described in "Configuring the classloader firewall" on page 583.

The forth section of the CE file defines the locations where the plug-in classloader searches for the resources it needs. This section is contained in the `ce:loader` element, which is also a child of `ce:environment`. The resource locations are specified in a `ce:location` element, a `ce:url` element, and two other elements as described in "Specifying the locations for the classloader" on page 584.

Chaining classloaders

You chain a CE by setting the `parent` attribute in the `ce:environment` element. The possible settings are:

- Attribute not set.

  If the parent attribute is not set, the classloader responsible for loading the bus is the parent of the plug-in's classloader firewall.

- `parent="`*`ParentCEName`*`"`

  The classloader whose name is *`ParentCEName`* is the parent of the plug-in's classloader firewall. If the specified classloader does not exist, the bus' classloader is used.

- `parent="system-classloader"`

  The system classloader is the parent of the plug-in's classloader firewall.

Configuring the classloader firewall

The classloader firewall assumes that all classes not specified by a positive filter are to be blocked from the Artix runtime's classloader. You define positive filters using one of the two `ce:filter` element's attributes: `type="discover"` and `type="pattern"`.

Using type="discover"

The discover filter type specifies that the classloader will discover the filters from the location specified in the `discover-source` attribute. Table 39 shows the values for `discover-source`.

Table 39: *discover-source values for the Classloader 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 firewall definition.
`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 resources. This option makes it possible to discover the contents of jar files that you know are reachable through the class loading system, but that 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 classloader. 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>.

Negative filters

Occasionally a positive filter will allow classes that you want blocked from the Artix runtime classloader to be visible through the firewall. This is particularly true with the package 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 classloader 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 locations for the classloader

The ce:loader element in the CE file specifies where the classloader will look for the resources it needs. These resources can be located on the local machine, on a networked machine, or even on the Web. You can specify their location using either pathnames or URLs.

To specify a resource's location using a pathname you use the ce:location element. Pathnames can be either absolute or relative. In addition they can include system variables. For example, the resource definition in Example 354 will use the value of LIB to resolve the specified path.

Example 354: *Resource Location Using a Variable*

```
<ce:location>$(LIB)\xml-apis.jar</ce:location>
```

To specify a resource's location using a URL you use the ce:url element. The classloader will use the URL to locate the classes specified.

In addition to ce:location and ce:url you can use two special elements to include resources:

ce:inherit-parent-locations specifies that the classloader will also use the resources defined in its parent classloader.

ce:tools-tar specifies that the current JDK's `tools.jar` is a resource for the classloader.

Example

Example 355 shows a sample CE file.

Example 355: *Simple CE File*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ce:classloader-environment PUBLIC "-//IONA//DTD IONA Classloading Environment 2.0//EN"
 "http://www.iona.com/dtds/classloader-environment_2_0.dtd">
<ce:classloader-environment xmlns:ce="http://www.iona.com/ns/classloader-environment"
 loglevel="info">

<ce:classloader-environment>
  <ce:environment name="sifter_ce">
    <ce:firewall>
      <ce:filter type="discover" discover-source="jre"/>
      <ce:filter type="negative-pattern">com.iona.jbus.jms.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.runtime.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.types.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.jaxrpc.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.ntv.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.util.</ce:filter>
      <ce:filter type="pattern">com.iona.jbus.</ce:filter>
      <ce:filter type="pattern">com.iona.jbus.servants.</ce:filter>
      <ce:filter type="pattern">com.iona.webservices.reflect.types.</ce:filter>
      <ce:filter type="pattern">com.iona.schemas.references</ce:filter>
      <ce:filter type="pattern">javax.xml.rpc.</ce:filter>
      <ce:filter type="pattern">javax.xml.namespace.QName</ce:filter>
    </ce:firewall>
    <ce:loader>
      <ce:location>/usr/iona/artix/lib/apache/jakarta-log4j/1.2.6/log4j.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/apache/xerces/2.5.0/xercesImpl.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/artix/java_runtime/3.0/it_bus.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/artix/ws_common/3.0/it_wsdl.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/artix/ws_common/3.0/saaj-api.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/artix/ws_common/3.0/it_saaj.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/artix/ws_common/3.0/it_ws_reflect.jar</ce:location>
      <ce:location>/usr/iona/artix/lib/common/ifc/1.1/ifc.jar</ce:location>
    </ce:loader>
  </ce:environment>
</ce:classloader-environment>
```

Configuring your applications

To configure the plug-ins in your application to use the Artix classloader environment you need to modify the application's configuration scope in the Artix configuration file, `artix.cfg`. For each plug-in that will use the Artix classloader environment you need to add two configuration variables:

plugins:*`plugin_name`***:CE_Name** specifies the name of the classloader that the plug-in specified will use to load. The CE name is defined in the classloader's configuration file.

ce:*`ce_name`***:FileName** specifies the name of the classloader's configuration file. *`ce_name`* must match the name specified in the plug-in's CE name configuration.

For example, if your application loads a plug-in called `sifter` that uses the Artix classloader environment and the classloader environment is configured using a file called `sifter_ce.xml`, then your application's configuration would look similar to Example 356.

Example 356:*Configuring a Plug-In to use the Classloader Environment*

```
#artix.cfg
pluginApp
{
1 orb_plugins=[...,"java"];
  java_plugins=["sifter"];
  plugins:sifter:classname="sifterFactory";
2 plugins:sifter:CE_Name="sifter_ce";
3 ce:sifter_ce:FileName="..\etc\sifter_ce.xml";
  ...
}
```

The entries in Example 356 do the following:

1. Configures the application to load the Java plug-in `sifter`.
2. Specifies that sifter uses a classloader environment named `sifter_ce`.
3. Specifies that the file defining `sifter_ce` is located at `..\etc\sifter_ce.xml`.

For more information on configuring Artix applications to use plug-ins see "Configuring Artix Plug-Ins" on page 563 and Deploying and Managing Artix Applications.

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 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 replies. For more details see "Working with Artix Message Contexts" on page 282.

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 "Using Artix References" on page 221.

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 Service 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 90.

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 54.

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 Simple Types by Restriction" on page 68.

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 231.

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 "Using Exceptions" on page 159.

H

Handler

`Hanlder` is the Java interface that a developer must implement to create a handler. It has has methods for processing both request and response messages. Artix provides a `GenericHandler` class to provide a template for implemeting handlers. See also "Writing Handlers" on page 479.

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 an `xsd:list` element. For more details see "Using Lists" on page 77.

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 handler that processes messages between the Artix binding to the Artix transport. See "Writing Handlers" on page 479.

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 267.

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, 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 handler that processes messages between the Artix binding and the user's application code. See "Writing Handlers" on page 479.

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 Bus

The infrastructure that allows service providers and service consumers to interact in a distributed environment. Handles the delivery of messages between different middleware systems. Also known as an Enterprise Service Bus.

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 clones 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 37.

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 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 38.

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 element

See Binding and Payload format.

WSDL fault element

See Fault message.

WSDL message element

See Message.

WSDL operation element

See Operation.

WSDL port element

See Service.

WSDL portType element

See Interface.

WSDL service element

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

D

E

O

P

R

S

T

U

V

W

X