PROGRESS SOFTWARE

Artix® ESB

Developing Artix Applications with JAX-RPC

Version 5.5, December 2008

Updated: May 29, 2009

Contents

Part II Advanced Artix Programming

List of Figures

List of Tables

Preface

What is Covered in this Book

This guide discusses the main aspects of developing transport-independent services and service consumers using JAX-RPC style stubs and skeletons generated by Artix ESB. 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

This guide 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 ESB to develop JAX-RPC 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 consumer, Chapter 2 describes the steps to implement a consumer using Artix-generated code.

If you are interested in the basics of writing an Artix-enabled service, Chapter 3 describes the steps to implement a service using Artix-generated code. It also includes details about the threading models used by Artix services.

Chapter 4 and Chapter 5 extend the discussion of building Artix applications. They discuss methods for discovering Artix contracts, getting access to an Artix bus, 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 7 gives detailed description of how all of the XML Schema 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 8 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.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see Using the Artix Library.

Part I

Fundamentals of Artix Programming

In this part

This part contains the following chapters:

CHAPTER 1

The Artix Java Development Model

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

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 dictating the type of messaging system over which the application works. 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 a collection of operations exposed by an endpoint, 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 XML Schema. 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 service skeletons and consumer 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.

In this section

This section discusses the following topics:

Generating Java Code

Overview

The Artix development tools include a utility to generate service skeleton and consumer 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 119.

Generating code using Artix Designer

Artix Designer includes a full Java IDE and can generate Artix starting point code for you. These capabilities combined with Artix Designer's WSDL editing capabilities, make it an end-to-end service development tool.

To generate Artix code inside Artix Designer need to do the following:

1. Create a launch configuration for your service.
2. Run the code generator from the **Artix Tools** dialog.

If you make changes to the contract from which your code is generated, you can regenerate the starting point code. Artix Designer will preserve any work you have done to the code. So, if you have implemented one of the operations in your contract and then add a new logical operation to the contract, you can regenerate the code. Your implementation code will be preserved and the starting point code for the new operation will be added.

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. For more information see Artix for J2EE.
`-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 499.
`-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 221 and Bindings and Transports, C++ Runtime.
`-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 Configuring and Deploying Solutions, C++ Runtime.
`-nexclude` *`namespace[=package]`*	Instructs the code generator to skip the specified XML Schema 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 XML Schema 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.

Mapping Contract Elements to Java

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 119.

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;
}
```

Java Package Naming

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.

CHAPTER 2

Developing Artix Consumers

Artix generates stub code that provides a developer with a simple model to develop consumers that can interact with services over a number of protocols.

In this chapter

This chapter discusses the following topics:

Generating the Stub Code

Overview

The Artix Java code generator generates the stub code needed to develop a consumer from an Artix contract. In addition, the code generator creates Java classes for the complex types defined in the contract using the mapping described in the JAX-RPC specification.

Generating code from the command line

You generate consumer code at the command line using the following command:

```
wsdltojava -client artix-contract
```

You must specify the location of a valid Artix contract for the code generator to work. The `-client` flag tells the code generator to generate the classes needed to develop a consumer from the specified contract.

Optional flags

You can also supply the following optional parameters to control what code is 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.
`-call`	Generates a consumer the uses the `Call` interface to invoke on the remote service. For more information see "Using the Call Interface for Dynamic Invocations" on page 499.
`-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 221 and Bindings and Transports, C++ Runtime.
`-merge`	Merge any user changes into the generated code.
`-nexclude namespace[=package]`	Instructs the code generator to skip the specified XML Schema 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 XML Schema 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`).
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.

Generated files

The Artix code generator produces the following files from when you generate code for a consumer:

- *portTypeName*`.java` defines the Java interface that the consumer's service proxy implements.
- *portTypeName*`TypeFactory.java` defines the type factories used by Artix to support the complex types used by the service.
- *portTypeName*`Client.java` is a simple main class for a consumer.

In addition, the code generator creates a class for each named schema type defined in the Artix contract. For more information on using these generated data types see "Working with Artix Data Types" on page 119.

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.

Writing the Consumer Code

Overview

Artix consumers 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 consumer initializes and shuts down the Artix bus.

An Artix consumer needs to do four basic things:

1. Initialize an instance of the Artix bus.
2. Instantiate one or more service proxies.
3. Invoke one or more operations on the service proxies.
4. Shut-down the Artix bus instance used by the consumer.

In this section

This section discusses the following topics:

Initializing an Artix Bus

Overview

The Artix bus manages the service proxy used to contact remote services. It also manages the invocation of any handlers used by the consumer. Your consumer code must initialize an instance of the Artix bus before you can register a service proxy and starting making requests on a remote service.

Bus.init()

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

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

`Bus.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 service proxy, load the Artix configuration information for your application, and load the required plug-ins. Once the bus is initialized, you can create a service proxy and register it with the bus. The bus will then take any invocations made on the service proxy and turn them into requests on the remote service.

Example

Example 7 shows code for initializing an instance of the Artix bus.

Example 7: *Initializing an Artix Bus*

```
public class HelloWorldClient
{

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

Creating a Service Proxy Using the JAX-RPC Method

Overview

Artix consumers use dynamic proxies, as described in the JAX-RPC specification, to make requests on remote services. 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 the proxy will use to 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`.

   Note: If your consumer is going to run inside of a J2EE container you will need to set the JAX-RPC `ServiceFactory` property to use the Artix `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` object for the service to which the proxy will connect.
3. Use the `Service` object to instantiate the dynamic proxy.

Obtaining a ServiceFactory instance

To obtain an instance of the `ServiceFactory` you call `ServiceFactory.newInstance()` as shown in Example 8. This returns the `ServiceFactory`. Only one is created per application and the same `ServiceFactory` is returned for each successive call.

Example 8: *Getting the ServiceFactory*

```
ServiceFactory factory = ServiceFactory.newInstance();
```

Creating a Service object

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

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

Example 9 shows an example of creating a `Service` object for a widget order service.

Example 9: *Creating a Service Object*

```
QName name = new QName("http://widgetVendor.com/widgetOrders",
                       "orderWidgetsService");

String wsdlPath = "http://widgetVendor.com/widgets.wsdl";
URL wsdlLocation = new File(wsdlPath).toURL();

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

Creating the dynamic proxy

The dynamic proxy is created using the `Service` objects' `getPort()` method. `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`.

As shown in Example 10, `getPort()` returns an instance of `java.rmi.Remote` that must be cast to the generated interface.

Example 10: *Creating the Dynamic Proxy*

```
QName portName = new QName("","orderWidgetsPort");

WidgetOrder proxy = (WidgetOrder)service.getPort(portName,
                                                 WidgetOrder.class);
```

Creating a Service Proxy Using Artix APIs

Overview

While the Artix Java APIs 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 11.

Example 11: *Bus.createClient()*

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

Example

Example 12 shows the code for creating a service proxy using `createClient()`.

Example 12: *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 12 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.

Shutting Down the Artix Bus

Overview

The Artix bus created to host a consumer's service proxy and handle the marshalling of requests and responses uses a number of resources. To ensure that all of the resources allocated by the bus is cleaned up, the bus needs to be properly shut down before the consumer application exits.

Bus.shutdown()

You shutdown a bus using its `shutdown()` method. This method takes one boolean argument that determines how the method returns control to the calling object. If you pass in `true`, `shutdown()` will block until the bus's internal threads have finished processing all requests and have fully shutdown. If you pass in `false`, `shutdown()` returns immediately. It is advisable to pass `true` to `shutdown()` to ensure that the bus is fully shutdown before exiting.

Example

Example 13 shows code for initializing an instance of the Artix bus.

Example 13: *Shutting Down an Artix Bus*

```
public class HelloWorldClient
{

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

Full Consumer Code

The code

An Artix consumer developed to access `HelloWorldService` will look similar to Example 14.

Example 14: *HelloWorld Consumer 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);
```

Example 14: *HelloWorld Consumer Code*

```
9       bus.shutdown(true);

        }
}
```

The explanation

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 through 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.

Setting 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 15 shows code getting a `Stub` object from an Artix proxy.

Example 15: *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 15 on page 48.
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 16 on page 49 shows code for setting the username for a client.

Example 16: *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 15 on page 48.
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 17 on page 50 shows code for setting the password for a client.

Example 17: *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 consumer 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 consumer's `Stub` interface do the following:

1. Get a `Stub` object by casting your service proxy to a `Stub` as shown in Example 15 on page 48.
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.ENDPOINT_ADDRESS_PROPERTY` as the property name and the `String` created in step 2 as the value of the property.

Example 17 on page 50 shows code for setting the endpoint address property.

Example 18: *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_ADDRESS_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 19. The syntax adheres to the rules for URL syntax described in RFC2396 (Uniform Resource Identifiers (URI): Generic Syntax).

Example 19: *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 20 shows a complete UDDI URL.

Example 20: *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 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 21 shows an example of creating a service proxy using UDDI.

Example 21: *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 21 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 your 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 22 shows a configuration fragment with the configuration to use UDDI.

Example 22: *UDDI Configuration*

```
collieClient

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

For more information on configuring Artix see Configuring and Deploying Solutions, C++ Runtime.

Note: The UDDI plug-in is implented using a JNI layer in the Artix ESB C++ Runtime.

Building an Artix Consumer

Required jar files

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

- *InstallDir*\artix_5.1\cxx_java\lib\artix\java_runtime\5.0\it_bus-api.jar
- *InstallDir*\artix_5.1\cxx_java\lib\ws_common\wsdl\1.3\it_wsdl.jar
- *InstallDir*\artix_5.1\cxx_java\lib\ws_common\reflect\1.3\it_ws_reflect.jar
- *InstallDir*\artix_5.1\cxx_java\lib\ws_common\reflect\1.3\it_ws_reflect_types.jar
- *InstallDir*\artix_5.1\cxx_java\lib\common\ifc\1.3\ifc.jar
- *InstallDir*\artix_5.1\cxx_java\lib\jaxrpc\jaxrpc\1.1\jaxrpc-api.jar

Other jar files

If your consumer uses SOAP with attachments, you will also need to include *InstallDir*\artix_5.1\cxx_java\lib\sun\activation\1.0.2\activation.jar on your classpath.

If your consumer uses `xsd:any`, you will need to include *InstallDir*\artix_5.1\cxx_java\lib\sun\saaj\1.2.1\saaj-api.jar on your classpath.

CHAPTER 3

Developing Artix Services

Artix generates the starting point code needed to develop and deploy protocol agnostic services.

Overview

Developing a service with Artix is a two step process The first step is to implement the business logic for your service. Because Artix generates JAX-RPC compliant code from your contracts, the implementation of your service's business logic does not require much Artix specific knowledge. Most of the code used will be standard Java code and manipulating the objects generated to handle complex types. Artix does have a number of proprietary APIs that are used to support some of its more advanced features.

The second step in developing an Artix service is to develop the code that registers your service's implementation with the Artix bus. This step involves some knowledge of Artix and how you intend to deploy your service. Artix provides you with two models for developing and deploying a service:

- The Artix container model
- Standalone model

The Artix container model is the preferred method. When using the container model you package your service as a plug-in that is deployed into a light-weight Artix container. The Artix container can host and manage a

number of services that use the same configuration scope. It provides a remote management APIs for dynamically starting and stopping your services.

The standalone deployment model requires that you develop your service as a standalone Java application. You can also develop your standalone application to host multiple services. However, this requires you to write the Java code for this and to ensure that your application cleans up it resources properly. The standalone model also does not provide the remote management APIs.

In this chapter

This chapter discusses the following topics:

Generating the Skeleton Code

Overview

The Artix development tools take an Artix contract and generate skeleton code to use as a starting point for developing a service. In addition, Artix maps WSDL types to Java classes using the mapping described in the JAX-RPC specification.

Generating code from the command line

You generate service skeleton code at the command line using the command:

```
wsdltojava -impl -plugin -deployable artix-contract
```

You must specify the location of a valid Artix contract for the code generator to work. The command line flags do the following:

- `-impl` instructs the code generator to create an empty implentation class for the service.
- `-plugin` instructs the code generator to create the plug-in classes needed to deploy the service into an Artix container.
- `-deployable` instructs the code generator to create a deployment descriptor for deploying the service into an Artix container.

Optional parameters

You can also supply the following optional parameters to control the generated code:

`-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.
`-server`	Generates a simple main class for the server. This flag is used in place of `-plugin` and `-deployable`. For more information see "Developing a Standalone Service" on page 75.
`-types`	Generates the code to implement the complex types defined by the contract.
`-interface`	Generates the Java interface for the service.
`-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 221 and Bindings and Transports, C++ Runtime.
`-merge`	Merge any user changes into the generated code.
`-nexclude namespace[=package]`	Instructs the code generator to skip the specified XML Schema 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 XML Schema 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`).
`-quiet`	Specifies that the tool runs in quiet mode.
`-verbose`	Specifies that the tool runs in verbose mode.

Generated files

The Artix code generator produces the following files from when you generate code for a service:

- *portTypeName*`.java` defines the Java interface that both the service implements.
- *portTypeName*`Impl.java` defines the class used to implement the service.
- *portTypeName*`ServicePlugin` includes code to register the appropriate servant with the bus when the service is loaded into an Artix container.
- *portTypeName*`ServicePluginFactory` instantiates the generated plug-in class for your service.
- *portTypeName*`TypeFactory.java` defines the type factories used by Artix to support the complex types used by the service.

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 119.

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.

Developing a Service Implementation

Generating the server implementation class

The Artix code generation utility, `wsdltojava`, will generate an implementation class for your service 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 service implementation class code consists of two files:

PortName.**java** contains the interface implemented by the service.

*PortName***Impl.java** contains the class definition for the service'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 service. 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 23. Only the code portions highlighted in **`bold`** (in the bodies of the `greetMe()` and `sayHi()` methods) must be inserted by the programmer.

Example 23: *Implementation of the HelloWorld PortType in the Server*

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

Example 23: *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";
    }
```

Developing a Container Based Service

Overview

The recommended method for deploying Artix services is to use the Artix container. This enables you to dynamically deploy, start, and stop your services using the container's remote interfaces. It also means that you can deploy multiple services into a single process.

Note: All of the services deployed into an Artix container will share one configuration scope.

To enable your services to be deployable into an Artix container you need to do the following:

1. Generate the starting point code for your service.
2. Implement the `busInit()` and `busShutdown()` methods in the generated plug-in class.
3. Implement the `activateService()` and `deactivateService()` methods in the generated servant activator class.
4. Complete the deployment descriptor for your service.

In this section

This section discusses the following topics:

Generating Starting Point Code

Overview

The `wsdltojava` tool, or the Artix Designer, will generate all the code you need to deploy your service into an Artix container. The generated classes provide basic implementations for all of the required methods. However, you may wish to modify the code for more advanced applications.

The wsdltojava flags

The `-plugin` and `-deployable` flags instruct the wsdltojava tool will generate the starting point code for a service to deploy into an Artix container. The `-plugin` flag instructs the code generator to generate the following additional classes:

- *portTypeName*`ServicePlugin` contains the code used by the bus when it loads the service plug-in.
- *portTypeName*`ServicePluginFactory` contains the code to instantiate the generated plug-in class for your service.
- *portTypeName*`ServiceActivator` contains the code to activate and deactivate the service.

The `-deployable` flag instructs the code generator to generate a deployment descriptor named *portTypeName*`.xml`. The deployment descriptor is used by the Artix container to load your service.

The generated implementations

The generated code provides a default implementation for all of the required methods. The default implementation includes the code needed to register and activate a single instance of a single service. It does not perform any resource initialization beyond the creation of the service instance.

If you want your service plug-in to load multiple services you will need to modify the `busInit()` method of the generated plug-in class. You may also need to add additional code to initialize and clean up any resources needed by the services loaded by the plug-n.

Implementing the Service's Plug-in Class

Overview

All Artix plug-ins have two classes. The first class, *portTypeName*`ServicePluginFactory`, is a factory used by Artix to create instances of the plug-in as needed. This class is fully implemented when you generate Artix starting point code for a container deployed service. You do not need to edit it. For more information see "Implementing the BusPlugInFactory Interface" on page 534

The second class, *portTypeName*`ServicePlugin`, is used by Artix to load your service's implementation, register it with the Artix bus, and instantiate any resources needed by the service. It is also used by the bus at shutdown to clean up any resources used by your service. The generated implementation of this class is sufficient for most services, however you may need to modify it.

When modifying the plug-in you will change two methods:

- busInit() is called by Artix when the plug-in is loaded. It is where you instantiate a servant for your service and register it with the bus.
- busShutdown() is called by Artix at shutdown. It is where you clean up any resources used by your service.

Implementing busInit()

`busInit()` is responsible for loading all the resources needed by service. This includes creating and registering the service activator that loads and unload the servant that hosts the service's implementaiton. To implement `busInit()` for your service you need to do four things:

1. Get an instance of the Artix bus.
2. Create an instance of your service's activator class.
3. Register the service's activator with the bus.
4. Call `activateService()` to load the servant for your service.

Getting an instance of the Artix bus

In order to register your servant with Artix, you need an instance of the Artix bus. The `BusPlugin` class, which your service's plug-in extends, has a method, `getBus()`, that returns the instance of the bus loading the plug-in. `getBus()` takes no arguments and returns a `Bus` object.

Instantiating a service activator

The creator for the generated service activator takes two arguments:

- the location of the service's contract.
- the bus that hosts the plug-in.

The default implementation of `busInit()` passes the hard coded location of the contract used to generate the service plug-in. Using a hard coded location for a contract limits the flexibility of your service plug-in. You should update your plug-in to use one of the methods outlined in "Finding Artix Contracts" on page 93.

Registering a service activator

You register a service activator with the bus using the bus' `registerServiceActivator()` method. The signature for `registerServiceActivator()` is shown in Example 24.

Example 24: *registerServiceActivator()*

```
public abstract boolean registerServiceActivator(QName serviceName, ServiceActivator sa)
throws BusException;
```

The `QName` passed into `registerServiceActivator()` is used by the bus to determine when to use this particular service activator object. It should be the same QName as that used to register the servants.

For more information on service activators see "Implementing the Service's Activator Class" on page 70.

Calling activateService()

`activateService()` is a method implemented by the service activator. It is responsible for instantiating a servant for your service and registering the servant with the bus. For more information on `activateService()` see "Implementing the Service's Activator Class" on page 70.

Example

Example 25 shows a `busInit()` method used in implementing the `SOAPService` service to be deployed in an Artix container.

Example 25: *busInit()*

```
import java.net.URL;
import javax.xml.namespace.QName;

import com.iona.jbus.Bus;
import com.iona.jbus.ServiceActivator;
import com.iona.jbus.BusConstants;
import com.iona.jbus.BusException;
import com.iona.jbus.BusPlugIn;

public class SOAPServicePlugin extends BusPlugIn
{
  private ServiceActivator serviceActivator;

  ...

  public void busInit() throws BusException
  {
    Bus bus = getBus();

    QName serviceName = new
   QName("http://www.iona.com/hello_world_soap_http",
          "SOAPService");
    String wsdl = bus.getServiceWSDL(serviceName);

    serviceActivator = new SOAPServiceServiceActivator(wsdl,
                                                       bus);

    bus.registerServiceActivator(serviceName, serviceActivator);

    serviceActivator.activateService(serviceName);
  }
  ...
}
```

Implementing busShutdown()

`busShutdown()` is called by Artix when the service is stopped or the Artix container is shutdown. It is where you would place code to clean up after your service. Typically `busShutdown()` needs to perform two tasks:

1. Call `deactivateService()` on the service's activator to clean up the servant used by the service.
2. Call `dereigisterServiceActivator()` to remove the service activator from the bus' registry.

Example 26 shows the default implementation of `busShutdown()` for the `SOAPService` plug-in.

Example 26: *busShutdown()*

```
public void busShutdown() throws BusException
{
  Bus bus = getBus();

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

  serviceActivator.deactivateService(serviceName);

  bus.deregisterServiceActivator(serviceName);
}
```

Implementing the Service's Activator Class

Overview

The service activator class provides the entry point for creating and registering servants. In general, this class is used to manage the lifecycle of an Artix service. If the relevant member functions of the service activator class are properly implemented, it should be possible to deactivate and then re-activate a service without needing to shut down the entire service plug-in.

Service activator functions

The service plug-in class provides two methods that control the lifecycle of an Artix service, as follows:

- `activateService()`—a method called either from within `busInit()` or whenever the `it_container_admin -deploy` command is executed.

  The purpose of `activateService()` is to perform all of the housekeeping tasks necessary to start up an Artix service, including the creation of a servant object and the registration of that servant object with the bus.
- `deactivateService()`—a method called either from within `busShutdown()` or whenever the `it_container_admin -removeservice` command is executed.

  The purpose of `deactivateService()` is to perform all of the housekeeping tasks necessary to shut down an Artix service, including deregistration of the service and deletion of the associated servant object.

Related container administration commands

The lifecycle functions provided by the service activator class are closely related to the following `it_container_admin` commands:

- `it_container_admin -deploy`—the effect of issuing this command depends on whether this is the first or subsequent deployment, as follows:
  - *First deployment*—load and initialize the service plug-in. The container calls `busInit()`, which is normally programmed to call `activateService()` for each of the WSDL services.

- *Subsequent deployment (re-deploy)*—activate any inactive services. The container calls `activateService()` on each of the registered service activators, but only if the service is currently inactive. The container does not call `busInit()` in this case.

Note: Artix does not currently provide an administration command that re-activates a single service at a time. The `-deploy` command re-activates all of the inactive services from the specified plug-in.

- `it_container_admin -removeservice`—de-activate a specific service. When you issue the `-removeservice` command, the container calls `deactivateService()`, but only if the specified service is currently active.

For more details about the `it_container_admin` command-line utility, see Configuring and Deploying Artix Solutions, C++ Runtime.

Note: JAX-RPC based services run in the Artix ESB C++ Runtime using a JNI layer.

---

activateService()

`activateService()` is called either from `busInit()` or whenever the `it_container_admin -deploy` command is issued. It is the appropriate place to put the code that creates and registers servants. Registering a servant is a two step process:

1. Create a servant for your service.
2. Register the service with the bus.

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 27. The creator for a `SingleInstanceServant` uses the following three items:

- the path of the WSDL file describing the service interface
- an instance of your implementation object
- an instance of an initialized Artix bus.

Example 27 shows the code to create a servant for the `HelloWorld` service.

Example 27: *Creating a Servant*

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

For more details see "Servant Threading Models" on page 83.

Registering a servant

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 service. The signature for `registerServant()` is shown in Example 28.

Example 28: *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 service's contract. 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.

For more details about servant registration see "Servant Registration" on page 79.

Example

Example 29 shows an implementation of `activateService()` that registers a Greeter servant, thereby associating it with the `SOAPService` WSDL service.

Example 29: *Sample Implementation of activate_service()*

```
public void activateService(QName serviceName)
throws BusException
{
  Servant servant = new SingleInstanceServant(new GreeterImpl(),
                                              theWsdlLocation,
                                              theBus);
  theBus.registerServant(servant, serviceName);
}
```

In this example, it is assumed that the service activator instance was registered as shown in Example 25 on page 68—that is, the service activator instance is registered *only* against the `SOAPService` WSDL service. Hence, it follows that the `activateService()` method shown in Example 29 will only be called when `serviceName` equals the `SOAPService` QName.

Advanced applications might choose to register a service activator instance against several different services. In that case, you would need to examine the service QName, `serviceName`, in order to decide which servant to activate.

---

deactivateService()

`deactivateService()` is called either by `busShutdown()` or whenever the `it_container_admin -removeservice` command is issued. It is the appropriate place to deregister the servant for your service. This is done by using the bus' `removeServant()` method.

Example 30 shows an implementation of `deactivateService()` that deregisters and deletes the Greeter servant that was registered by `activateService()`.

Example 30: *Sample Implementation of deactivate_service()*

```
public void deactivateService(QName serviceName)
{
  try
  {
    theBus.removeServant(serviceName);
  }
  catch(BusException ex) {}
}
```

Developing a Standalone Service

Overview

If you decide that you want to deploy your service as a standalone Java application, Artix can generate a server class that contains a `main()`. Developing a service as a standalone service requires that you register your service implementation, or implementations, with the Artix bus in the applications `main()`. It also requires that you must specifically initialize the Artix bus and then start the bus.

Your standalone service will require a dedicated configuration scope. However, it will not require a deployment descriptor.

Generating the service main()

You can use `wsdltojava` to generate a service `main()` by using the `-server` flag as shown in Example 31.

Example 31: *Generating a Standalone service*

```
wsdltojava -server -impl widgets.wsdl
```

The `-server` flag is used in place of the `-plugin` and the `-deployable` flags. It instructs the code generator to create a class containing a `main()`. The generated `main()` will contain the basic code needed to register the service implementation with the bus. The `main()` shown in Example 34 on page 77 was generated using `wsdltojava`.

Writing the main()

The `main()` of a standalone service must do four things before it can process 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.

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 service, load the Artix configuration information for your application, and load the required plug-ins.

Before the bus can begin processing requests made on your service, you must register the servant object that implements your service'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 as it processes 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 32. The creator for a `SingleInstanceServant` uses the following:

- an instance of your implementation object
- the path of the WSDL file describing the service interface
- an instance of an initialized Artix bus.

Example 32 shows the code to create a servant for the `HelloWorld` service.

Example 32: *Creating a Servant*

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

For more details see "Servant Threading Models" on page 83.

Registering a servant

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 service. The signature for `registerServant()` is shown in Example 33.

Example 33: *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 service's contract. 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.

For more details about servant registration see "Servant Registration" on page 79

Starting the bus

After the bus is initialized and the service implementation is registered with it, the bus is ready to listen for requests and pass them to the servant 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 service's `main()` will be blocked until `run()` returns.

Completed server main()

Example 34 shows the `main()` for a standalone service.

Example 34: *Server main()*

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

public class Server
{
  public static void main(String args[])
  throws Exception
  {
```

Example 34: *Server main()*

```
    // Initialize the Artix bus
    Bus bus = Bus.init(args);

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

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

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 `service` 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 the details of a `service` element appearing in the service's 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 consumers 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 33.

Example 35: *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 specify a different port name on each call.

Example

Example 36 shows the code for registering a static servant.

Example 36: *Registering a Static Servant*

```
QName name = new QName("http://whoDunIt.com/Sleuth",
                       "SleuthService");
Servant servant = new SingleInstanceServant("./sleuth.wsdl",
                                            new SleuthImpl());
bus.registerServant(servant, name, "SleuthHTTPPort");
```

Transient Servant Registration

Overview

When a servant is registered as a *transient* servant, Artix clones a `service` element from the service's physical contract 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 QName is replaced by a dynamically generated, unique 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 such 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 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 `service` element 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 `service` elements defined in the contract. If you place your service templates after your actual `service` elements, you may run into problems using the router.
- The `port` elements defined in the `service` element must use one of the supported transports.
- The `port` elements defined in the `service` element must fully describe the properties of the transport being used.
- The address specified for an HTTP endpoint must be specified using *`host_name`*`:0`.

- The address specified for either a CORBA endpoint or a IIOP endpoint 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 37.

Example 37: *registerTransientServant()*

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

In addition to the servant instance, `registerTransientServant()` takes the `service` element's QName as specified in the contract defining the service. Unlike `registerServant()`, `registerTransientServant()` does not allow you to specify a `port` element 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` element 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 38 shows the code for registering a transient servant.

Example 38: *Registering a Transient Servant*

```
QName name = new QName("http://whoDunIt.com/Sleuth",
                       "SleuthService");
Servant servant = new SingleInstanceServant("./sleuth.wsdl",
                                            new SleuthImpl());
QName transientName = bus.registerTransientServant(servant,
                                                   name);
```

Servant Threading Models

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 Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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

endpoint access the same implementation object as shown in Figure 1 on page 84. The `SingleInstanceServant` provides no thread safety for the user code.

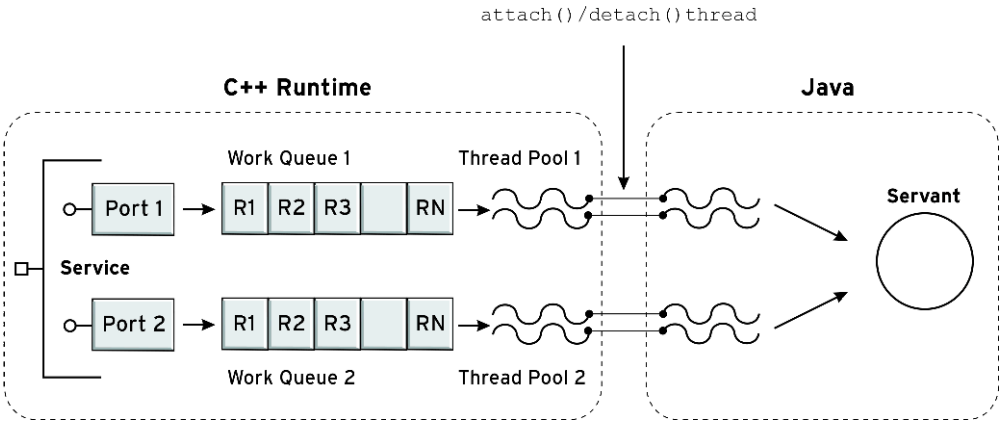

Figure 1: *SingleInstanceServant*

To instantiate a `SingleInstanceServant` you need to provide an instance of your implementation object, the path of the contract describing the service, and an instance of an initialized Artix bus. Example 32 shows an example of instantiating a `SingleInstanceServant`.

Example 39: *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 85. Using a `SerializedServant` is equivalent to using a `SingleInstanceServant` whose target object is completely synchronized.

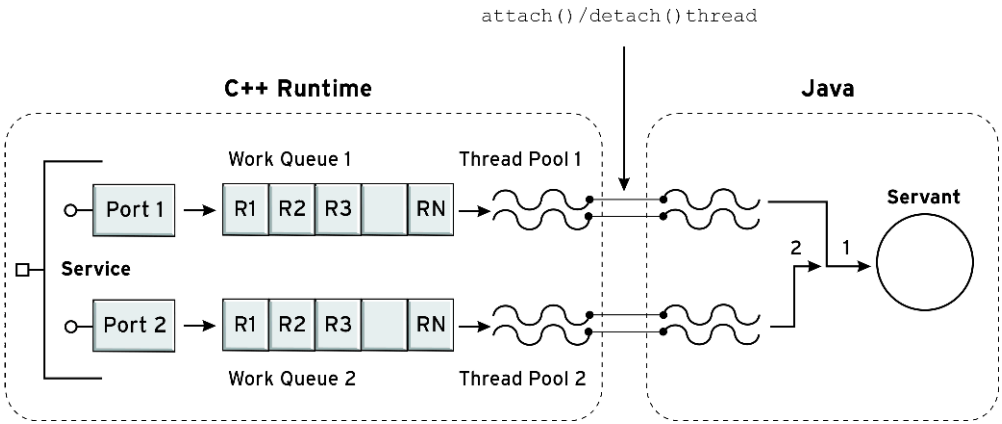

Figure 2: *SerializedServant*

To instantiate a `SerializedServant` you need to provide an instance of your implementation object, the path of the contract describing the service, and an instance of an initialized Artix bus. Example 32 shows an example of instantiating a `SerializedServant`.

Example 40: *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 86. This ensures thread safety, but does not allow the implementation object to have any statefull information.

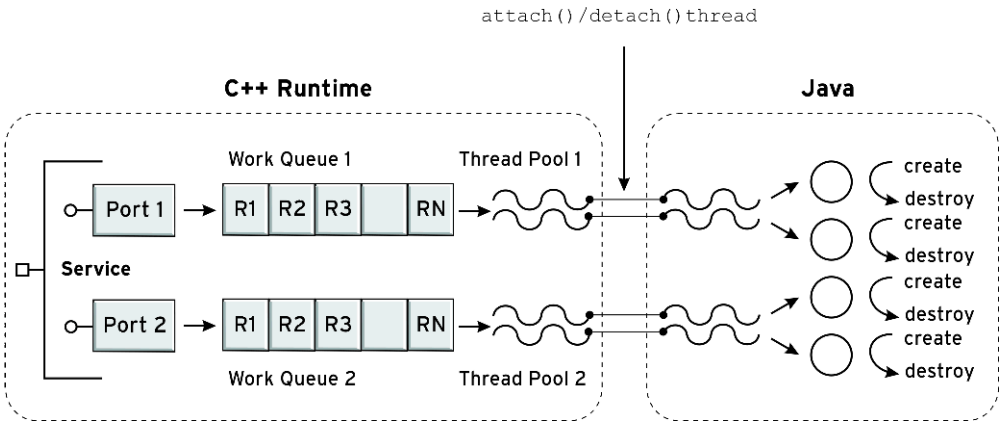

Figure 3: *PerInvocationServant*

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 an instance of your implementation object, the path of the contract describing the service, and an instance of an initialized Artix bus when being instantiated. Example 41 shows the code for instantiating a `PerInvocationServant`.

Example 41: *Creating a PerInvocationServant*

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

Building an Artix Service

Required jar files

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

- *InstallDir*\artix_5.1\cxx_java\lib\artix\java_runtime\5.0\it_bus-api.jar
- *InstallDir*\artix_5.1\cxx_java\lib\ws_common\wsdl\1.3\it_wsdl.jar
- *InstallDir*\artix_5.1\cxx_java\lib\ws_common\reflect\1.3\it_ws_reflect.jar
- *InstallDir*\artix_5.1\cxx_java\lib\ws_common\reflect\1.3\it_ws_reflect_types.jar
- *InstallDir*\artix_5.1\cxx_java\lib\common\ifc\1.3\ifc.jar
- *InstallDir*\artix_5.1\cxx_java\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*\artix_5.1\cxx_java\lib\sun\activation\1.0.2\activation.jar on your classpath.

If your application uses `xsd:any`, you will need to include *InstallDir*\artix_5.1\cxx_java\lib\sun\saaj\1.2.1\saaj-api.jar on your classpath.

CHAPTER 4

Finding Contracts and References at Runtime

Locating contracts at runtime is much more flexible than specifying their location at development time.

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.

Artix simplifies the process of obtaining the following kinds of basic resources: 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 chapter

This chapter discusses the following topics:

Finding Initial References

Overview

An endpoint 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 application has a reference to a service, it creates a service proxy by passing the reference to a proxy constructor.

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

Note: The Artix 3.0.x API `Bus.resolveInitialReference()` has been deprecated in Artix 4.0. It is supported for backwards compatibility, but it is recommended that you update clients to use the newer API.

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 42.

Example 42: *Finding an Initial Reference*

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

EndpointReferenceType ref;

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

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

proxy.sayHi();
```

Options for finding initial references

Artix finds initial references from the following sources, in order of priority:

1. *Collocated service*—if the client code that calls `resolveInitialEndpointReference()` is in the same process as the specified service, `resolveInitialEndpointReference()` returns an endpoint 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 XML instance of an endpoint 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 endpoint reference file or by specifying the literal value of an endpoint reference.

   For more details, see Configuring and Deploying Artix Solutions, C++ Runtime.

   Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

4. *Service in a contract*—the `service` element in a contract contains essentially the same data as an endpoint reference. Hence, if an endpoint 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 discovering references is, thus, effectively an extension of the mechanism for discovering service contracts—see "Options for finding contracts" on page 93.

Finding Artix 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 discovered by Artix.

The Artix provides an API, `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 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 43.

Example 43: *Finding a Contract*

```
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 finding contracts

Artix 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 Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implented on top of the Artix ESB C++ Runtime using a JNI layer.

3. *Contract directory specified on the command line*—you can provide a contract by specifying a contract directory on the command line. When Artix 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 Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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*—Artix 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 79.

For more information on endpoint references see "Using Endpoint References" on page 281.

CHAPTER 5

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:

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 44.

Example 44: *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 45.

Example 45: *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();
```

Ensuring a Server Uses a Unique Bus

Overview

The default behavior of Artix is to create a single `Bus` instance per JVM. This means that if you are deploying multiple Artix service providers into a single servlet container, a single application server, a single Artix container, or a single Artix server, all the services will use the same `Bus` instance. For all cases except multiple service providers in an Artix container, you can ensure that each service provider gets a unique `Bus` instance. This is done by providing a unique ORBId to the `init()` method used to instantiate the service providers `Bus` instance.

Note: There is no way to spawn multiple `Bus` instances in side of an Artix container.

You can specify an ORBId in one of two ways:

- Specifying the **`-ORBId`** parameter on the command line when starting an Artix server.
- Adding a `-ORBId` entry to the argument list passed to `Bus.init()` when creating a new `Bus` instance for a service provider.

Specifying the ORBId on the command line

When starting an Artix Java server from the command line you can supply a number of optional command line parameters. These parameters are used to specify configuration information to the `Bus` instance started by the server.

Among these optional parameters is the **`-ORBId`** parameter. This parameter specifies the ORB identifier used when creating the `Bus` instance used by an Artix server.

When it is added to the command line, as shown in Example 46, the **`ORBId`** parameter instructs the Artix runtime to associate the server with a `Bus` instance using the specifed ORB identifier. If an appropraite `Bus` instance does not exist in the JVM, a new `Bus` instance is created for the server. If a `Bus` instance with the specified ORB identifier exists, the JVM will return that `Bus` instance.

Example 46: *Starting an Artix Server with the ORBId Parameter*

```
java com.iona.demo.HelloWorldServer -ORBId HelloBus
```

Specifying the ORBId programatically

You can specify the ORBId of the `Bus` instance used by an Artix server by adding the `-ORBId` parameter to the array of strings passed to the `Bus.init()` method. Example 47 shows code for doing this.

Example 47: *Specifying the Bus's ORBId Programatically*

```
public class HelloWorldServer
{
  public static void main(String args[])
  {
1   String[] newArgs = new String(args.length + 2);

2   int i = 0;
    while (i < args.length)
    {
      newArgs[i] = args[i];
      i++;
    }

3   newArgs[i] = "-ORBid";
    newArgs[i+1] = "HelloORB";

4   Bus bus = Bus.init(newArgs);

    ...
  }
}
```

The code in Example 47 does the following:

1. Creates a new array to hold the updated command line argument list.
2. Copies the command line arguments into the new argument list.
3. Adds the `-ORBId` parameter to the new argument list.
4. Calls `Bus.init()` with the updated argument list.

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.

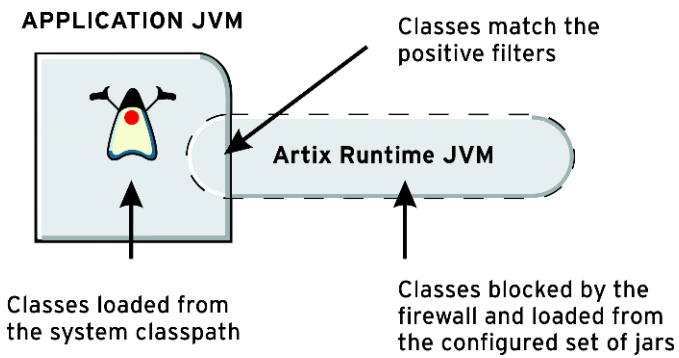

Figure 4: *Classloader Firewall*

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 48 to specify the proper Xerces and Log4J version to load into the Artix runtime.

Example 48: *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.

Avoid Circular References

Overview

This section warns against using circular references in Artix code. These can result in infinite recursion and stack overflow.

Circular references and infinite recursion

Artix does not detect circular references in your code. You must ensure that your code does not contain any circular references.

The following simple example class contains a single member object:

```
class A {
    Object m;
}
```

The following code shows an example circular reference:

```
A myObject = new A();
myObject.m = myObject;
```

In this example, Artix first marshals the `myObject` instance, and then marshals members of the `myObject` instance, which in this case is `myObject.m`. This leads back to the `myObject` instance, which is marshalled again, followed again by the `myObject.m` member, resulting in an infinite recursive loop.

Artix does not check for circular references for performance reasons. Circular references can result in infinite recursion and stack overflow and must be avoided.

CHAPTER 6

Handling Artix Generated 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:

Generic Exception Handling

Overview

By default, remote invocations in Java return a `RemoteException` when the remote service throws an exception. This works fine when working with other Java services. Artix, however, is designed to interact with services developed on a number of platforms. It is unlikely that user defined exceptions and `RemoteException` objects can cover all of the possible exceptions.

To fix this limitation, Artix uses a class called `com.iona.jbus.FaultException` to handle exceptions thrown by remote endpoints.

Bindings and Artix exceptions

Each binding supported by Artix handles Artix generated exceptions differently. Some, such as SOAP and CORBA, have mappings that are determined by standards. Others use proprietary mappings. Table 2 describes how each of the bindings handle Artix generated exceptions.

Table 2: *Binding Support for Artix Exceptions*

Binding	Support
SOAP	Artix runtime exceptions and user thrown `FaultException` objects are mapped into `SOAPFaultException` objects. For more information see "Using the SOAP Binding" on page 116.
CORBA	Artix runtime exceptions and user thrown `FaultException` objects are mapping into corresponding CORBA exceptions. For more information see Artix for CORBA.
Fixed	Artix runtime exceptions and user thrown `FaultException` objects are mapped into a fixed record length message using a proprietary mapping.
Tagged	Artix runtime exceptions and user thrown `FaultException` objects are mapped into a tagged message using a proprietary mapping.

Table 2: *Binding Support for Artix Exceptions*

Binding	Support
TibMsg	Artix runtime exceptions and user thrown `FaultException` objects are mapped into a `TibMsg` using a proprietary mapping.
FML	
XML	Artix runtime exceptions and user thrown `FaultException` objects are not transmitted.
G2++	Artix runtime exceptions and user thrown `FaultException` objects are not transmitted.

When working with bindings that use proprietary mappings for exceptions, Artix will transmit a message containing the exception back to the remote endpoint. If the remote endpoint is developed using Artix, it will properly decode the exception and behave as described in this section. If it is not developed using Artix, it is responsible for decoding the message being returned.

In this section

This section discusses the following topics:

Overview of Fault Exceptions

Overview

`FaultException` inherits from `RuntimeException` and adds fields to hold the information needed to support the range of exceptions that Artix can encounter. 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.

FaultException fields

`FaultException` objects have four fields. These field are explained in Table 3.

Table 3: *FaultException Fields*

Name	Description
Message	Specifies a detailed description of why the exception was thrown.
Category	Specifies the category of the exception. For a full listing of the possible fault categories see the FaultCatagory Javadoc.
Completion Status	Specifies the status of the invocation. For a full listing of the possible values see the FaultCompletionStatus Javadoc.
Source	Specifies the type of endpoint that threw the exception. For a full listing of the possible values see the FaultSource Javadoc.

Artix runtime exceptions

The Artix runtime has a number of implementation specific exception types that can be thrown. Artix runtime exceptions that occur along the messaging chain are not passed to the user code. Instead they are packaged into a `FaultException` and passed back down the message chain. The binding level and transport level code will package the exception into an appropriate format and transmit it back to the remote endpoint.

Some Artix runtime exceptions are returned to the user-level code. You must handle these exceptions. One method of handling them is to throw a user defined exception as discussed in "Creating User-Defined Exceptions" on page 229. Alternatively, you can throw your own `FaultException` as discussed in "Throwing Fault Exceptions" on page 114.

Processing Fault Exceptions

Overview

In general, your applications will not catch `FaultException`. For local method calls, they should catch the exceptions that are thrown by the local method. For remote method calls, they should catch `RemoteException` as specified by JAX-RPC. The `FaultException` is stored in the cause member of the caught `RemoteException`.

Procedure

To extract the `FaultException` do the following:

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

Getting exception details from a FaultException

`FaultException` objects have four getter methods, shown in Example 49, to retrieve the information about the cause of the exception.

Example 49: *FaultException Getter Methods*

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

Evaluating the exception data

The values returned by three of the methods are instances of an enumeration. 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 50.

Example 50: *Evaluating the Completion Status of a Fault Exception*

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

Example

Example 51 shows code for catching and inspecting a `FaultException`.

Example 51: *Catching a FaultException*

```
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() == (FaultCategory.TRANSIENT_VAL))
    {
      // a TRANSIENT system exception
    }
```

Example 51: *Catching a FaultException*

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

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

Throwing Fault Exceptions

Throwing a FaultException

Because `FaultException` extends `RuntimeException`, you can throw a `FaultException` just as you would any other exception in your application code. The Artix runtime will process the exception and populate the message according to the binding and transport being used by the endpoint. If the endpoint receiving the exception is an Artix endpoint, it will interpret the `FaultException` and return it to the endpoint's application logic as a `RemoteException`. If the receiving endpoint is not an Artix endpoint, it will need to have logic for interpreting the fault message that is transmitted.

Procedure

To throw a FaultException from your code do the following:

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 52, takes a single string that is placed in the message field of the new object.

Example 52: *FaultException Constructor*

```
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 53, to populate the fields used to report details about the exception.

Example 53: *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 54.

Example 54: *Setting the Source Field*

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

Example

Example 55 shows code for throwing a fault exception from an Artix service.

Example 55: *Throwing a Fault Exception*

```
FaultException fe = new FaultException("Account has expired");
fe.setCategory(FaultCategory.TIMEOUT);
fe.setSource(FaultSource.SERVER);
fe.setCompletionStatus(FaultCompletionStatus.NO);
throw fe;
```

Using the SOAP Binding

Overview

According to the JAX-RPC specification, exceptions are mapped to `soap:fault` elements when using the SOAP binding and `soap:fault` elements are mapped to either a `RemoteException`, a user defined exception, or a `SOAPFaultException`. Artix runtime exceptions and user thrown `FaultException` objects are mapped to `SOAPFaultException` objects.

Catching exceptions

When using the SOAP binding, Artix applications need catch `SOAPFaultException` objects. When a remote invocation results in a returned exception, the Artix SOAP binding will either return a user defined exception or a `javax.xml.rpc.soap.SOAPFaultException` object. If the remote endpoint is implemented using Artix, a `SOAPFaultException` is returned when either:

- an Artix runtime exception occurred.
- the application code threw a `FaultException` object.

You can inspect the `SOAPFaultException` object's `FaultString` field to determine the cause of the exception. It contains the `String` from the `Message` field of the `FaultException` that caused the `SOAPFaultException`.

Example 56 shows code for catching a `SOAPFaultException` and inspecting its `FaultString` field.

Example 56: *Catching a SOAPFaultException*

```
try
{
  String returnVal = impl.sayHi();
  System.out.println("Returned: "+returnVal);
}
catch(SOAPFaultException sfe)
{
  System.out.println("Caught exception");
  System.out.println("Fault String: "+sfe.getFaultString());
}
```

Throwing exceptions

When throwing exceptions from Artix applications using the SOAP binding, you do not need to do anything special. You can throw a `FaultException` object and the SOAP binding will map it into a `SOAPFaultException`. In the mapping the `FaultException` object's `Message` field is mapped to the `SOAPFaultException` object's `FaultString` field.

You can also throw `SOAPFaultException` object directly.

More information

For more information on `SOAPFaultException` objects see the `SOAPFaultException` Javadoc.

CHAPTER 7

Working with Artix Data Types

Artix maps XML Schema data types in an Artix contract into Java data types. For XML Schema 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:

Including and Importing Schema Definitions

Overview

Artix supports the including and importing of schema definitions, using the `<include/>` and `<import/>` schema tags. These tags enable you to insert definitions from external files or resources into the scope of a `schema` element. The essential difference including and importing is this:

- Including brings in definitions that belong to the *same* target namespace as the enclosing `schema` element, whereas
- Importing brings in definitions that belong to a *different* target namespace from the enclosing `schema` element.

xsd:include syntax

The include directive has the following syntax:

```
<include schemaLocation = "anyURI" />
```

The referenced schema, given by *anyURI*, must either belong to the same target namespace as the enclosing schema or not belong to any target namespace at all. If the referenced schema does not belong to any target namespace, it is automatically adopted into the enclosing schema's namespace when it is included.

xsd:import syntax

The import directive has the following syntax:

```
<import namespace = "namespaceAnyURI"
        schemaLocation = "schemaAnyURI" />
```

The imported definitions must belong to the *namespaceAnyURI* target namespace. If *namespaceAnyURI* is blank or remains unspecified, the imported schema definitions are unqualified.

Example

Example 57 shows an example of an XML schema that includes another XML schema.

Example 57: *Example of a Schema that Includes Another Schema*

```
<definitions
   targetNamespace="http://schemas.iona.com/tests/schema_parser"
    xmlns:tns="http://schemas.iona.com/tests/schema_parser"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
        <schema
  targetNamespace="http://schemas.iona.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">

            <include schemaLocation="included.xsd"/>

            <complexType name="IncludingSequence">
                <sequence>
                    <element
                        name="includedSeq"
                        type="tns:IncludedSequence"/>
                </sequence>
            </complexType>

        </schema>
    </types>
<...>
```

Example 58 shows the contents of the included schema file, `included.xsd`.

Example 58: *Example of an Included Schema*

```
<schema
  targetNamespace="http://schemas.iona.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
    <!-- Included type definitions -->
    <complexType name="IncludedSequence">
        <sequence>
            <element name="varInt" type="int"/>
            <element name="varString" type="string"/>
        </sequence>
    </complexType>
</schema>
```

XML Schema Elements

Schema elements

Elements in XML Schema represent an instance of an element in an XML document generated from the schema. At their most basic, an element consists of a single `element` element. 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 XML Schema 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 XML Schema 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 XML Schema 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 XML Schema Simple Types

Overview

Artix follows the JAX-RPC specification for mapping native XML Schema types into Java. In most cases, the mapping from an atomic XML Schema 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 XML Schema types, the generated parameter's type will be of a corresponding primitive Java type. For example, the message description shown in Example 59 will cause a parameter, `score`, of type `int` to be generated.

Example 59: *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 4.

Table 4: *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
xsd:unsignedByte	byte

Table 4: *Simple Schema Type to Primitive Java Type Mapping*

Schema Type	Java Type
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 4: *Simple Schema Type to Primitive Java Type Mapping*

Schema Type	Java Type
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 4: *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 XML Schema 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 XML Schema atomics types you should take care to ensure that they conform to the restrictions of the XML Schema 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.

BigDecimal type

In Artix, the `java.math.BigDecimal` type is stored internally as a `jstring`. The recommended way in Artix to initialize an object of `BigDecimal` type with a floating-point number is as follows:

```
new BigDecimal(Double.toString(645.769));
```

Special Atomics Type Mappings

Overview

Mapping XML Schema atomic types to Java primitives does not work for all possible data descriptions in an Artix contract. Several cases require that an XML Schema 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 60.

  Example 60: *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 61.

  Example 61: *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 62.

  Example 62: *Optional Attribute Description*

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

Mappings

Table 5 shows how XML Schema simple types are mapped into Java wrapper classes in these special cases.

Table 5: *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
xsd:duration	java.lang.String

Defining Simple Types by Restriction

Overview

XML Schema 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 XML Schema 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 XML Schema

Example 63 shows the syntax for describing a simple type.

Example 63: *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 64 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 64: *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 64, would be mapped to a `String` because the base type of `SSN` is `xsd:string`. For example, the contract fragment shown in Example 65 would result in a Java method, `creditInfo()`, which took a parameter, `socNum`, of `String`.

Example 65: *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 should ensure that your application logic enforces the restrictions described in the contract for maximum interoperability.

Enforced facets

For the facets that Artix does enforce, no special code is generated. Instead, the enforcement is done by the Artix core. Therefore, the Artix user level code will allow you to set invalid values into a restricted simple type. However, when the Artix core attempts to parse the message, it will throw a runtime exception and refuse to process the message.

Artix enforces the following facets:

length

The `length` facet is a non-negative integer that works with a number of primitive types. Table 6 describes the effects of the `length` facet on supported XML Schema types.

Table 6: *Effects of length Facet on XML Schema Types*

Restricted Type	Effect
`xsd:string`	The string must have the specified number of characters.
`xsd:anyURL`	The URL must have the specified number of characters.
`xsd:list`	The list must have the specified number of elements.
`xsd:hexBinary`	The value must have the specified number of octets (8-bits).
`xsd:base64Binary`	The value must have the specified number of octets (8-bits).

minLength

The `minLength` facet is a non-negative integer that works with a number of primitive types. Table 7 describes the effects of the `minLength` facet on supported XML Schema types.

Table 7: *Effects of minLength Facet on XML Schema Types*

Restricted Type	Effect
`xsd:string`	The string must have at least the specified number of characters.
`xsd:anyURL`	The URL must have at least the specified number of characters.
`xsd:list`	The list must have at least the specified number of elements.
`xsd:hexBinary`	The value must have at least the specified number of octets (8-bits).
`xsd:base64Binary`	The value must have at least the specified number of octets (8-bits).

maxLength

The `maxLength` facet is a non-negative integer that works with a number of primitive types. Table 8 describes the effects of the `maxLength` facet on supported XML Schema types.

Table 8: *Effects of maxLength Facet on XML Schema Types*

Restricted Type	Effect
`xsd:string`	The string must have no more than the specified number of characters.
`xsd:anyURL`	The URL must have no more than the specified number of characters.
`xsd:list`	The list must have no more than the specified number of elements.
`xsd:hexBinary`	The value must have no more than the specified number of octets (8-bits).

Table 8: *Effects of maxLength Facet on XML Schema Types*

Restricted Type	Effect
`xsd:base64Binary`	The value must have no more than the specified number of octets (8-bits).

enumeration

For more information on the enumeration facet, read "Using Enumerations" on page 135.

Unenforced facets

Artix does not enforce the following facets:

- `pattern`
- `whiteSpace`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`
- `totalDigits`
- `fractionDigits`

Using Enumerations

Overview

In XML Schema, enumerations are described by derivation of a simple type using the syntax shown in Example 66.

Example 66: *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 67, would be valid if it were `<widgetSize>big</widgetSize>`, but not if it were `<widgetSize>big,mungo</widgetSize>`.

Example 67: *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 175.

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 68, to represent the enumeration, `widgetSize`, shown in Example 67 on page 135.

Example 68: *WidgetSize Class*

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

Example 68: *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 68: *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 68: *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 67 on page 135, 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 69 shows a simple text entry method for getting the proper member of the enumeration using `fromValue()`,

Example 69: *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 70 uses `getValue()` to return the value of the enumeration retrieved in Example 69 on page 139

Example 70: *Using getValue()*

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

Using Lists

Overview

XML Schema 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 71.

Example 71: *List Type Example*

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

In Java code list types are mapped into arrays.

Defining list types in XML Schema

XML Schema 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 72.

Example 72: *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 XML Schema 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 9 shows the facets used by list types.

Table 9: *List Type Facets*

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

Table 9: *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 71 on page 141, is shown in Example 73.

Example 73: *Definition for simpleList*

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

In addition to the syntax shown in Example 72 on page 141 you can also define a list type using the less common syntax shown in Example 74.

Example 74: *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 75 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 75: *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 76, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 76: *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 XML Schema Unions

Overview

In XML Schema, 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`.

XML Schema 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 77 shows the definition of a union that can store either an integer or a string.

Example 77: *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 78 shows an example of a union containing an anonymous member type restricting the possible values of a valid integer to `1` through `10`.

Example 78: *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 135

The Java mapping of XML Schema unions is very similar to that used in mapping choice complex types. See "Choice Complex Types" on page 154. 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 77 on page 144, would result in the generated class shown in Example 79.

Example 79: *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 79: *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 XML Schema 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 XML Schema 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 XML Schema Complex Types

Overview

Complex types are described in the `types` section of an Artix contract. Typically, they are described in XML Schema 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. XML Schema 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 80, the generated setter method would have the signature `void setName(String val)`.

Example 80: *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 80 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 81 would result in the generation of a private variable, `observedSpeed`,of type `float[]`.

Example 81: *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 82.

Example 82: *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 83.

Example 83: *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 84.

Example 84: *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 84: *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

XML Schema 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 85.

Example 85: *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 86.

Example 86: *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 86: *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 86: *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 XML Schema `currency` would be described using an `attribute` element as shown in Example 87 on page 159.

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 90 on page 160.

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 XML Schema

An XML Schema `attribute` element has one required attribute, `name`, that is used to identify the attribute. It also has four optional attributes:

`use`	Specifies if the attribute is `required`, `optional`, or `prohibited`.
`type`	Specifies the type of value the attribute can take. If it is not used the schema type of the attribute must be defined in-line.
`default`	Specifies a default value to use for the attribute. It is only used when the attribute definition's `use` attribute is set to `optional`.
`fixed`	Specifies a fixed value to use for the attribute. It is only used when the attribute definition's `use` attribute is set to `optional`.

Example 87 shows an `attribute` element defining an attribute, `currency`, whose value is a string.

Example 87: *XML Schema 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 must be described in-line. Example 88 shows an `attribute` element for an attribute, `catagory`, that can take the values `autobiography`, `non-fiction`, or `fiction`.

Example 88: *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 89 shows an alternate description of the `catagory` attribute using the `type` attribute.

Example 89: *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>
```

Describing an attribute group in XML Schema

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 XML Schema" on page 158. Example 90 shows the description of the attribute group `catalogIndecies`. The attribute group has two members. `catagory` is of the type defined in Example 89 on page 160. `pubDate` is of the native XML Schema type `dateTime` and is required.

Example 90: *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 91.

Example 91: *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="catalogIndices" />
</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 92 would generate a class with three sets of getter/setter methods.

Example 92: *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 93.

Example 93: *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 93: *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 91, 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 94

Example 94: *dvdType Java Class*

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

Example 94: *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 94: *dvdType Java Class*

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

Undeclared Attributes

Overview

XML Schema has a mechanism that allows you to leave a place holder for an arbitrary attribute in a complex type definition. Using this mechanism, you could define a complex type that can have any attribute. For example, you could create a type that defines the elements `<robot name="epsilon" />`, `<robot age="10000" />`, or `<robot type="weevil" />` without specifying the three attributes. This can be particularly useful when you need to provide for a bit of flexibility in your data.

Defining in XML Schema

Undeclared attributes are defined in XML Schema using the `anyAttribute` element. It can be used wherever an `attribute` element can be used. The `anyAttribute` element has no attributes as shown in Example 95.

Example 95: *Complex Type with an Undeclared Attribute*

```
<complexType name="arbitter">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="rate" type="xsd:float" />
  </sequence>
  <anyAttribute />
</complexType>
```

The defined type, `arbitter`, has two elements and can have one attribute of any type. The elements `<officer rank="12"><name>...</name><rate>...</rate></officer>`, `<lawyer type="divorce"><name>...</name><rate>...</rate></lawyer>`, and `<judge><name>...</name><rate>...</rate></judge>` can all be generated from the complex type `arbitter`.

Note: The `anyAttribute` element is not support for complex types with an `all` element.

Mapping to Java

When a complex type containing an `anyAttribute` element is mapped to Java, Artix adds a member called `otherAttributes` to the generated class. `otherAttributes` is of type `java.util.Map` and as with all other attributes it has a getter method and a setter method. Example 96 shows the class generated for the complex type defined in Example 95.

Example 96: *Class for a Complex Type with an Undeclared Attribute*

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

import java.util.Map;

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

  public static final javax.xml.namespace.QName QNAME = new
   javax.xml.namespace.QName("http://schemas.iona.com/types",
   "arbitter");

  private String name;
  private float rate;
  private Map otherAttributes;

  public String getName()
  {
    return name;
  }

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

  public float getRate()
  {
    return rate;
  }

  public void setRate(float val)
  {
    this.rate = val;
  }
```

Example 96: *Class for a Complex Type with an Undeclared Attribute*

```
  public Map getOtherAttributes()
  {
    return otherAttributes;
  }

  public void setOtherAttributes(Map val)
  {
    this.otherAttributes = val;
  }


  public javax.xml.namespace.QName _getQName()
  {
    return QNAME;
  }

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

Setting undeclared attributes

The `otherAttributes` member of the generated class expects to be populated with a `HashMap` object. The map is keyed using QNames. You can set the keys using either the standard `javax.xml.namespace.QName` object or the Artix specific `com.iona.common.util.QName` object. Once you have created and populated the hash map, you can set the `otherAttributes` member using the `setOtherAttributes()` method as shown in Example 97.

Example 97: *Setting Values for Undeclared Attributes*

```
judge = new Arbitter();

1 otherAtts = new HashMap();

2 QName at1 = new QName("test.iona.com", "house");
  QName at2 = new QName("test.iona.com", "veteran");

3 otherAtts.put(at1, "Cape");
  otherAtts.put(at2, "false");

4 judge.setOtherAttributes(otherAtts);
```

The code in Example 97 does the following:

1. Creates a new `HashMap` object to hold the undeclared attributes.
2. Creates `QName` objects for each of the attribute names.
3. Puts two attributes into the hash map.
4. Sets the `otherAttributes` member of the object using its `setOtherAttributes()` method.

Any changes to the hash map will be reflected in the value of the `otherAttributes` member once it is set.

Inspecting undeclared attributes

You retrieve the hash map holding undeclared attributes using the `getOtherAttributes()` method. `getOtherAttributes()` returns a Java `Map` object that is keyed using `com.iona.common.util.QName` object. Example 98 shows code for checking the value of an undeclared attribute.

Example 98: *Checking the Values for an Undeclared Attribute*

```
1  import com.iona.common.util.QName;

   // object judge populated earlier
2  Map otherAttrs = judge.getOtherAttributes();

3  QName atKey = new QName("test.iona.com", "house");

4  String houseType = (String)otherAttrs.get(atKey);
```

The code in Example 98 does the following:

1. Imports the Artix specific `QName` class.
2. Retrieves the `Map` object containing the undeclared attributes.
3. Creates a `QName` object for the desired attribute.
4. Gets the value from the hash map.

Nesting Complex Types

Overview

XML Schema 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 99.

Example 99: *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 99 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 99 on page 170 your application would include two classes to support it, `TweetyBird` and `SylvesterState`.

Example 100 shows the generated Java class for `tweetyBird`.

Example 100: *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 100: *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 101, has one element, `food`, that is an instance of `TweetyBird`.

Example 101: *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 101: *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 102 shows an example of using the nested type `sylvesterState` in Java.

Example 102: *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);
```

1

2

3

Example 102: *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 102 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 103 shows a description of `sylvesterState` using an anonymous type.

Example 103: *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 103 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 100 on page 171 your application would include the class `SylvesterState` to support it.

The generated class for `sylvesterState`, shown in Example 104, contains an internal class `SylvesterState.Food`. The element `food` is an instance of `SylvesterState.Food`.

Example 104: *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 104: *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 104: *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 102 shows an example of using the nested type `sylvesterState` in Java.

Example 105: *Working with Nested Complex Types*

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

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

hunter.setFood(prey);

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

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

The code in Example 102 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.

There are two ways of deriving a complex type from a simple type:

- by extension
- by restriction

Derivation by extension

Example 106 shows an example of a complex type, `internationalPrice`, derived by extension from the `xsd:decimal` simple type to include a currency attribute.

Example 106: *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`.

Derivation by restriction

Example 107 shows an example of a complex type, `idType`, that is derived by restriction from an `xsd:string`. The defined type must have a value that is ten characters in length. In addition, `idType` has an attribute called `expires`.

Example 107: *complexType derived from a simpleType using Restriction*

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
    </restriction>
    <attribute name="expires" type="xsd:dateTime" />
  </simpleContent>
</complexType>
```

As is Example 106 the `simpleContent` element signals that the new type does not contain any children. However, the definition uses a `restriction` element to constrain the possible values used in the new type. The `attribute` element adds the attribute to the new type.

Java mapping

A complex type derived from a simple type is mapped to a Java class. The generated 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.

When a complex type is derived by restriction the generated `set_value()` method will enforce the following facets:

- `length`
- `maxLength`
- `minLength`

If you attempt to set an invalid value, `set_value()` will throw a `RuntimeException`. For more information on the effects of the facets see X-REF.

Example

Example 108 shows the generated Java class representing the `idType` complex type from Example 107.

Example 108: *idType Java Class*

```
//Java
public class IdType
{

  public static final String TARGET_NAMESPACE = "tracking.gov";


  private String _value;
  private static final BigInteger length = new BigInteger("10");
  private Calendar expires;

  public String get_value()
  {
    return _value;
  }

  public void set_value(String val)
  {
    BigInteger realLength = new
   BigInteger(String.valueOf(val.length()));
   if (realLength.compareTo(length) == 0)
   {
     _value = val;
     return;
   }
   throw new RuntimeException("Invalid length value in
   org.soapinterop.xsd.IdType");
  }

  public Calendar getExpires()
  {
    return expires;
  }

  public void setExpires(Calendar val)
  {
    this.expires = val;
  }
```

Example 108: *idType Java Class*

```
  public javax.xml.namespace.QName _getQName()
  {
    return QNAME;
  }

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

The simple type value (that is, the value enclosed between the `<idType>` and `</idType>` tags) is accessed and modified by the `get_value()` and `set_value()` methods. The `set_value()` method, due to the inclusion of the `length` facet, checks to ensure that the value is the proper length. The value of the expires attribute is accessed and modified using the `getExpires()` and `setExpires()` methods.

Deriving a Complex Type from a Complex Type

Overview

Using XML Schema, 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 XML Schema.

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

Example 109: *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 109: *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 110 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 110: *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 110: *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 109 on page 184 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 111 shows the generated class for `widgetOrderBillInfo`.

Example 111: *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

XML Schema allows you to specify the occurrence constraints on three different XML Schema 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 114 shows the definition of a sequence type, `CultureInfo`, with sequence occurrence constraints. The sequence can be repeated 0 to 2 times.

Example 112: *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 113 shows an outline of the Java class generated for the type defined in Example 112.

Example 113: *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 113: *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 113: *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 113: *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 114 shows the definition of a choice type, `ClubEvent`, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 114: *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 115 shows an outline of the Java class generated for the type defined in Example 114.

Example 115: *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 115: *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 115: *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 115: *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 115: *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 116.

Example 116: *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 116, 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 213.

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 XML Schema 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 116 will be mapped to the Java class `HouseCat` shown in Example 117.

Example 117: *HouseCat Java Class*

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

Example 117: *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

XML Schema 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 XML Schema

You define a model group in XML Schema 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 10 shows how the choice of child element affects the behavior of the elements in the group.

Table 10: *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 118 shows a model group with three elements.

Example 118: *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 118 you would use `<group ref="tns:passenger" />` as shown in Example 119.

Example 119: *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 118 as shown in Example 120.

Example 120: *Instance of a Type with a Group*

```
<reservation>
```

Example 120: *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 118 on page 201. 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 121.

Example 121: *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 121: *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 XML Schema Complex Types" on page 148 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 119 on page 201, would include a member of type `Passenger` as shown in Example 122.

Example 122: *Type with a Group*

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

Example 122: *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 XML Schema any Elements

Overview

An XML Schema `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 123 shows the syntax for defining an element as an `any` in an Artix contract.

Example 123: *Syntax of an any*

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

Table 11 explains the details of the optional attributes.

Table 11: *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 11: *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 XML Schema definition available to validate the contents of the `any`, then it must be valid. If there is no XML Schema definition available, then validation is skipped.

Example 124 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 124: *Complex Type with an any*

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

Mapping to Java

XML Schema `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 125 shows the Java class generated for the complex type `wildCard`, shown in Example 124 on page 207.

Example 125: *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 125: *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 126 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 126: *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 127 shows code for extracting the data from an `any` containing a `houseCat`, defined in Example 116 on page 198.

Example 127: *Parsing the Contents of an any*

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

WildCard dataHolder;
```

Example 127: *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 127 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 128 shows the code for creating an `any` element containing the XML document `<houseCat><name>Joe</name><lives>12</lives></houseCat>`.

Example 128: *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 128 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 129.

Example 129: *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 130.

Example 130: *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 131 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 131: *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 132, would result in the generation a Java method `celebWasher()` that took a parameter, `badLang`, of type `String[]`.

Example 132: *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 133, the mapping of `celebWasher()` would take a parameter, `badLang`, of type `String[][]`.

Example 133: *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 134 shows an example of an operation that would require holder classes in the generated Java code.

Example 134: *Multiple Output Parts*

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

Example 134: *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 135 shows the resulting Java method signature.

Example 135: *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 136 shows how a server might implement the router method.

Example 136: *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 137 shows a client calling `router()` on a portal service.

Example 137: *Client Calling router()*

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

Example 137: *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 12 shows the mappings for the supported MIME types.

Table 12: *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 138 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 138: *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 138: *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 139.

Example 139: *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 138 on page 222 would result in the operation shown in Example 140 as opposed to the one shown in Example 139 on page 224.

Example 140: *SOAP Attachments Using DataHandler Objects*

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

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

Example 140: *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 XML Schema Constructs

Unsupported built-in types

The following XML Schema types are currently not supported by Artix:

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

Unsupported simpleType features

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

- The `final` attribute
- All facets except for enumeration
- Have an `xsd:list` child element

Unsupported complexType features

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

- The `mixed` attribute
- The `final` attribute
- The `block` attribute
- The `abstract` attribute
- `simpleContent` with `restriction`
- `complexContent` with `restriction`
- Having a choice complex type with a child of `xsd:all`
- Using `xsd:anyAttribute`

Unsupported features of xsd:element

The following attributes are not supported for `xsd:element`:

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

The following children are not supported for `xsd:element`:

- `xsd:unique`
- `xsd:key`
- `xsd:keyRef`

Unsupported attributes for xsd:attribute

The following attributes are not supported for `attribute`:

- `ref`
- `from`

Unsupported features of xsd:attributeGroup

You cannot use `xsd:anyAttribute` inside of `xsd:attributeGroup`.

Unsupported attributes of xsd:anyAttribute

The following attributes are not supported for `xsd:anyAttribute`:

- `namespace`
- `processContents`

Unsupported xsd: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 XML Schema elements

The following XML Schema elements are not supported:

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

id attribute

The `id` attribute is not supported by Artix.

CHAPTER 8

Creating User-Defined Exceptions

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

In this chapter

This chapter discusses the following topics:

Describing User-defined Exceptions in an Artix Contract

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 Bindings and Transports, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Describing the exception message

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

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

Example 141 shows a message description in an Artix contract.

Example 141: *Message Description*

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

For more information on describing a message in an Artix contract, read Bindings and Transports, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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 142 shows an operation description that uses the message described in Example 141 on page 231 as a user-defined exception.

Example 142: *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 142, `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 141 on page 231, 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 143, 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 143: *Complex Fault*

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

Example 143: *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 119.

Example

Example 144 shows the generated exception class for the fault message in Example 141 on page 231.

Example 144: *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 144: *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 145 shows how a server implementing the `getWidgets` operation, shown in Example 142 on page 231, might instantiate and throw a `NotEnoughInventory` exception.

Example 145: *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 146 shows how a client might catch and report the exception thrown by the server.

Example 146: *Catching a User-defined Exception*

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

Example 146: *Catching a User-defined Exception*

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

CHAPTER 9

Using Substitution Groups

XML Schema 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 XML Schema `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 147.

Example 147: *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 148 where the elements of the substitution group are of types derived from the head element's type.

Example 148: *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 148 on page 239, the `<part>` elements in Example 149 are valid.

Example 149: *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 150. Using this schema, a valid `review` element could contain either a `positiveComment` element or a `negativeComment` element, but not a `comment` element.

Example 150: *Abstract Head Definition*

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

Example 150: *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.

Following a similar pattern to the one used in generating 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 148 on page 239. A possible definition `widgetOrderInfo` is shown in Example 151.

Example 151: *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 152 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 152: *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 152: *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 152: *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 152, `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 153 that uses the substitution group defined in Example 148 on page 239.

Example 153: *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 154 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 154: *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 155.

Example 155: *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 155 is shown in Example 156. In this scenario, if the head element is declared abstract the methods supporting it would not be generated.

Example 156: *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 156: *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 156: *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 148 on page 239. The service offers two operations: `checkWidgets` and `placeWidgetOrder`. Example 157 shows the interface for the ordering service.

Example 157: *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 151 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` uses two a complex types containing the substituion group. 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 157. The Artix WSDL to Java code generator creates the implementation class shown in Example 158 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 158: *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 159.

Example 159: *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 159: *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 160.

Example 160: *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 161 shows client code for setting the `widget` member of `WidgetOrderInfo`.

Example 161: *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 161: *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 161: *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 logic 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 10

Working with Artix Type Factories

Artix uses generated type factories to support a number of advanced features including XML Schema 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 XML Schema `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 XML Schema types and XML Schema 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 XML Schema document that defines types used by your application and it will generate the classes and type factory for the defined types.

Each contract or XML Schema 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 XML Schema 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 XML Schema 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 98.
2. Instantiate the type factories you wish to register with the client proxy as shown in "Instantiating a type factory" on page 264.
3. Register the type factories using `registerTypeFactory()` on the `Bus` object as shown in "Registering a type factory" on page 265.

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

Example 162: *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 163 shows code registering a type factory.

Example 163: *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 164 shows code for returning the hash table of registered type factories.

Example 164: *Getting Hash Table of Registered Type Factories*

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

Example

Example 165 shows an example of registering two type factories, `packageDepotTypeFactory` and `widgetsTypeFactory`.

Example 165: *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 165: *Registering Type Factories*

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

The code in Example 165 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 166 a calling `getSupportedNamespaces()` on the generated type factory would return an array of strings containing a single entry: `http://packageTracking.com/packageTypes`.

Example 166: *WSDL Fragment*

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

Example 166: *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 167 shows code calling `getSupportedNamespaces()`.

Example 167: *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 166 on page 267 would cause a class called `PackageInfo` to be generated to support the XML Schema complex type `packageInfo`. Calling `getSchemaType()` on an

instance of `packageDepotTypeFactory`, as shown in Example 168, would return a QName whose local part is `packageInfo` and whose namespace URI is `http://packageTracking.com/packageTypes`.

Example 168: *getSchemaType()*

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

---

getJavaType()

`getJavaType()` returns the Java `Class` object generated to support the specified XML Schema type. It takes the QName of an XML Schema 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 XML Schema type and returns an instance of it.

For example, the code in Example 169 gets an instance of the generated `PackageInfo` object by passing `getJavaType()` the QName of the `packageInfo` XML Schema type defined in Example 166 on page 267.

Example 169: *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 169 does the following:

1. Creates the QName for the XML Schema type.
2. Calls `getJavaType()` on the type factory to get the `Class` object for the XML Schema 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 XML Schema element. It takes the QName of an XML Schema 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 XML Schema element and returns an instance of it.
getTypeResourceLocation()	`getTypeResourceLocation()` returns a string containing the location of the contract, or XML Schema document, for which the type factory was generated.

CHAPTER 11

Working with XML Schema anyTypes

The XML Schema anyType allows you to place a value of any valid XML Schema 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 XML Schema anyTypes

XML Schema anyType

The XML Schema `anyType` is the root type for all XML Schema 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 XML Schema 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 XML Schema 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 264.

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 276.

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

Table 13: *anyType Setter Methods for Primitive Types*

Method	Java Type	XML Schema 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 13: *anyType Setter Methods for Primitive Types*

Method	Java Type	XML Schema Type
setUInt()	long	uint
setULong()	BigInteger	ulong
setDecimal()	BigDecimal	decimal

Setting complex type data

You set complex data into any AnyType using setType(). setType() can be used in one of two ways. The first is to provide the QName of the XML Schema 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 170 shows code for storing a widgetSize in an anyType.

Example 170: *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 pass in null for the QName and the data value to store in the AnyType. When it encounters a null QName, Artix will determine the XML Schema 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 content's schema type when it transmits the data. However, the application that stores the value will have no way to determine the data type until it is used as part of a remote invocation. Example 171 shows code for storing a widgetSize in an anyType without providing its QName.

Example 171: *Storing Complex Data without a QName*

```
// Java
widgetSize size = widgetSize.big;
AnyType aT =new AnyType();
aT.setType(null, 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 172 gets the schema type of an `anyType` and prints it out to the console.

Example 172: *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 14 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 14: *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 173 shows an example of retrieving a `widgetSize` from an `anyType`.

Example 173: *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 174 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 174: *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 174: *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 174 on page 278 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 12

Using Endpoint References

An endpoint reference is a standardized means of representing handles to Artix service instances. Because they can be passed as message parts, endpoint references provide a convenient and flexible way of identifying and locating specific services.

In this chapter

This chapter discusses the following topics:

Introduction to Endpoint References

Overview

An *Endpoint Reference* is a Java object that encapsulates the addressing information for an endpoint defined in a WSDL contract. They are generated from the WS-Addressing endpoint reference schema type. Endpoint references in Artix have the following features:

- The encapsulate the information stored in a `wsdl:service` element.
- They can be passed as a parameter of an operation.
- They can be used to create service proxies for a service.
- They are the building blocks for the Artix locator and the Artix session manager.
- They are transport neutral. An endpoint reference can be used to represent any Artix service.

Note: In versions of Artix prior to 4.0, references were represented by the proprietary `Reference` type. The `Reference` type has been deprecated and replaced by WS-Addressing compliant endpoint references. For details of the issues involved in migrating, see "Migration Scenarios" on page 323.

In this section

This section discusses the following topics:

Endpoint Reference Basic Concepts

Overview

An endpoint reference is a Java object, derived from the XML Schema defined by the WS-Addressing standard. It contains all of the information needed to contact a deployed Artix endpoint. It lists the endpoint's address and contains a copy of the `service` element from the endpoint's contract. The data contained in the reference provides an Artix client process with the information needed to instantiate a service proxy to contact the referenced endpoint.

Using endpoint references provides you with the ability to generate servants on the fly and pass a client an endpoint 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 endpoint references to supply applications with pointers to the services which they are looking-up.

Contents of an endpoint reference

An endpoint reference encapsulates the following data:

- *Endpoint Address*—the addressing details needed to contact the endpoint expressed as a IRI.
- *Reference Parameters*—an optional list of properties used to connect to the endpoint.
- *Metadata*—a WSDL document containing the `service` element containing the endpoint's `port` element. Because Artix associates endpoint references with the `service` element of an Artix contract, the `service` element included in the endpoint reference may contain multiple `port` elements.

Note: The `service` element contained in the endpoint reference's metadata is derived from the `service` element in the endpoint's physical contract. If the endpoint reference is generated for a transient servant or for an endpoint whose `port` element contains a dynamic URL, the `service` element in the metadata will contain the live information.

The schema definition of a reference

Like all types in Artix, the reference is defined in XML Schema. The XML Schema defining a reference is located in the `schema` folder of your Artix Installation and is called `wsaddressing.xsd`. It can also be found on-line at http://www.w3.org/2005/08/addressing/ws-addr.xsd.

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

Java mapping of an endpoint reference

In Java an endpoint reference is mapped to a class called `com.iona.schemas.wsaddressing.EndpointReferenceType`. This class is provided in the libraries shipped with Artix. Applications that use endpoint references must import this class.

Endpoint references and the Artix router

When endpoint references are passed through the Artix router, the router creates a service proxy for each endpoint 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 `service` element was used as the template to create the servant. It chooses the first compatible `service` element it encounters in the router's contract. A compatible `service` element is one that uses the same `portType` element as the template used to create the transient servant.

If your contract contains a `service` element for a static service and a `service` element for use as a template for transient services and they both use the same `portType` element, the router will use the first one listed in the contract. If the static `service` element is first, the router will create a proxy that connects to the servant defined by that `service` element and not the transient servant 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 `service` elements. 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 cannot reap 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 Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Using Endpoint References in Artix Contracts

Overview

There are many cases where distributed applications need to exchange contact information. For example, an endpoint may need to register a callback object or a service may be acting as a factory for other services. In cases where contact information is being exchanged, you will need to include endpoint references in one or more of the logical messages defined in your service's contract.

To use endpoint references in a service contract do the following:

1. Define a prefix, typically `wsa`, for the WS-Addressing schema used to define endpoint references.
2. Import the WS-Addressing schema in the `types` element of your contract.
3. Use the `wsa:EndpointReferenceType` in any logical data units or logical messages that involve the exchange of contact information.

Defining the wsa prefix

You define namespace prefixes in a contract's `definitions` element. They are used as shorthand for full namespace declarations throughout the body of the contract. Commonly used prefixes include `xsd` for the namespace under which XML Schema elements are defined and `soap` for the namespace under which SOAP elements are defined.

To define the wsa prefix and associate it with the namespace under which the WS-Addressing `EndpointReferenceType` is defined you will need to add the line shown in Example 175 to your contract's definition element.

Example 175: *Defining the wsa Prefix*

```
<definintions ...
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  ... >
```

The full namespace under which the WS-Addressing elements are defined is `http://www.w3.org/2005/08/addressing`. Once this line is added to your contract you will be able to use elements defined in the namespace by prefixing the element name with `wsa:`.

Importing the schema

Before you can use the `wsa:EndpointReferenceType` in your contract, you need to import the XML Schema document defining it into your contract. You import XML Schema documents into your contract using an `import` element as shown in Example 176.

Example 176: *Importing the WS-Addressing Schema*

```
<import location="http://www.w3.org/2005/08/addressing/ws-addr.xsd"
        namespace="http://www.w3.org/2005/08/addressing" />
```

The value of the `location` attribute is the location of the actual XML Schema document being imported. For the WS-Addressing XML Schema document, the most portable location is from `http://www.w3.org/2005/08/addressing/ws-addr.xsd`. The value of the `namespace` attribute should match the value of the prefix declaration in the `definitions` element.

Using the endpoint reference in a message

Once you've imported the XML Schema document defining `wsa:EndpointReferenceType`, you can use endpoint references as units of data in your custom type definitions or as parts of a message. You add endpoint references to custom types by adding an element that is of `wsa:EndpointReferenceType` as shown in Example 177. The complex type `referenceHolder` has two elements. `serviceName` is of type `xsd:string`. `serviceEndpoint` is of `wsa:EndpointReferenceType` and is used to hold and endpoint reference.

Example 177: *Using an Endpoint Reference in a Custom Type*

```
<complexType name="referenceHolder">
  <sequence>
    <element name="serviceName" type="xsd:string" />
    <element name="serviceEndpoint"
             type="wsa:EndpointReferenceType" />
</complexType>
```

You can specify that an endpoint reference is part of a logical message directly in one or more of the `message` element of a contract. To do so you set the `type` attribute of one of the logical message's `part` elements to `wsa:EndpointReferenceType`. For example, the message defined in Example 178 consists of only an endpoint reference.

Example 178: *Using an Endpoint Reference in a Logical Message*

```
<message name="factoryRequest">
  <part name="endpointInfo" type="wsa:EndpointReference" />
</message>
```

Creating a NULL Endpoint Reference

Overview

There may be cases where you want to create a NULL endpoint reference. To do so you would instantiate an endpoint reference object and a URI attribute object. You would set the attribute object to a NULL address and then set the address of the endpoint reference to the NULL URI attribute.

Procedure

To create a NULL endpoint reference do the following:

1. Instantiate an `EndpointReferenceType` object.
2. Instantiate a `URI` object with a NULL address.

```
java.net.URI null_addr = new
  java.net.URI("http://www.w3.org/2005/08/addressing/none");
```

3. Instantiate an `AttributedURIType`.

```
com.iona.schemas.wsaddressing.AttributedURIType uri_null =
  new com.iona.schemas.wsaddressing.AttributedURIType();
```

4. Set the `AttributedURIType`'s value to the NULL `URI`.

```
uri_null.set_value(null_addr);
```

5. Set the `EndpointReferenceType`'s address field to the NULL `AttributedURIType`.

```
ref.setAddress(uri_null);
```

Example

Example 179 shows the code for creating a NULL endpoint reference.

Example 179: *Creating a NULL EPR*

```
EndpointReferenceType ref = new EndpointReferenceType();
java.net.URI null_addr = new
  java.net.URI("http://www.w3.org/2005/08/addressing/none");
AttributedURIType uri_null = new AttributedURIType();
uri_null.set_value(null_addr);
ref.setAddress(uri_null);
```

Creating Endpoint References for a Service

Overview

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

1. Get a handle to a bus as shown in "Getting a Bus" on page 98.
2. Register the servant with the bus.
3. Create an endpoint 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 180 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 180: *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 79.

Creating the endpoint reference

Once you have registered a service with the bus, you can create an endpoint reference for it using the `QName` returned from the servant registration method. Endpoint references are created using the bus' `createEndpointReference()` method. Example 181 shows the signature for `createEndpointReference()`.

Example 181: *createEndpointReference()*

```
//Java
EndpointReferenceType createEndpointReference(QName service);
```

The method takes in the `QName` of a registered service. For a static servant, the service's `QName` is the `QName` of the service from the WSDL contract. For a transient servant, the `QName` of the service is returned when you register the transient servant with the bus. Keeping track of the registered service's `QName` when using endpoint 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.

Note: It is recommended that when your application is creating endpoint references, it has the `wsdl_publish` plugin loaded. If it is not, WSDL location stored in the endpoint reference will be local to the application creating the reference.

Example

Example 182 shows the code for generating an endpoint reference for a static instance of the `Cling` service.

Example 182: *Creating an Endpoint Reference*

```
//Java
import com.iona.jbus.*
com.iona.schemas.wsaddressing.EndpointReferenceType;

// 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.registerTransientServant(servant, name, "ClingPort");

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

Instantiating Service Proxies Using an Endpoint Reference

Overview

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

Getting a bus

Typically, you will receive an endpoint 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 183.

Example 183: *Getting a Bus Reference Inside a Servant*

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

Creating a service

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

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

You create service proxy from an endpoint reference by calling `createClient()` on the servant's default bus. `createClient()` takes an endpoint 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 184.

Example 184: *Bus.createClient()*

```
Remote Bus.createClient(EndpointReferenceType epr,
                        Class interfaceClass)
throws BusException
```

Example

Example 185 shows the code for creating a service proxy for the `Cling` service from an endpoint reference.

Example 185: *Creating a Service Proxy from and Endpoint Reference*

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

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

Using Endpoint References in a Factory Pattern

Overview

A common pattern for working with endpoint references is a factory pattern where one object, a factory, creates endpoint 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 endpoint 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 contract defining the `Bank` service has several key elements that are required for defining a service that uses endpoint references in a factory pattern. The first thing to notice is that the contract imports the XML Schema definition for endpoint references. Also, it defines two interfaces: `Bank` and `Account`. `Bank` defines an operation for returning endpoint references to an `Account`. Also, both interfaces have fully described bindings and service definitions.

Messages with endpoint references

The `Bank` interface's `get_account` operation returns an endpoint reference to an `Account`. The message definition for the response of these operations have one part, `return`, that is of type `wsa:EndpointReferenceType`. Example 186 shows the definition for a message that contains an endpoint reference.

Example 186: *Message with a Reference*

```
<message name="bankResponse">
  <part name="return" type="wsa:EndpointReferenceType" />
</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 an endpoint reference. Example 187 shows the `portType` element for the `Bank` interface.

Example 187: *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 contract 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 endpoint 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 endpoint reference into a generic CORBA `Object`. You can do some additional work to create typed CORBA references. For details on how endpoint references are mapped into a CORBA binding see Artix for CORBA.

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

Example 188: *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:wsa="http://www.w3.org/2005/08/addressing"
  xmlns:bank="http://www.iona.com/bus/demos/bank"
  targetNamespace="http://www.iona.com/bus/demos/bank"
  name="BankService">
  <import location="http://www.w3.org/2005/08/addressing/ws-addr.xsd"
          namespace="http://www.w3.org/2005/08/addressing" />
  <message name="accountName">
    <part name="account_name" type="xsd:string"/>
  </message>
  <message name="bankResponse">
    <part name="return" type="wsa:EndpointReferenceType"/>
  </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>
```

Example 188: *Bank Service Contract*

```
<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>
<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>
```

Example 188: *Bank Service Contract*

```
  <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 endpoint references to accounts. `get_account` creates a servant for an account and registers it as a transient servant. The accounts are registered as transient servants to ensure that each new account is exposed as a unique endpoint.

The bank service interface

The bank service defined in the contract shown in Example 188 will result in a generated interface called `Bank`. This interface extends `java.rmi.Remote` and is used by clients to create proxies for the bank service. While you generally do not need to edit this generated interface, you do need to edit it when using endpoint references. You will need to add a line to the generated interface that imports the `EndpointReferenceType` class as shown by the bold line in Example 189.

Example 189: *Bank Interface*

```
import java.rmi.RemoteException;

import com.iona.schemas.wsaddressing.EndpointReferenceType;
public interface Bank extends java.rmi.Remote
{
  public EndpointReferenceType get_account(String account_name)
  throws RemoteException;
}
```

The bank service implementation object

The bank service defined in the contract shown in Example 188 will result in a generated implementation class called `BankImpl`. This object will contain one method, `get_account()`, for which you will provide the logic. You will also need to add an import for the `EndpointReferenceType` class. The import statement is in bold to show that you need to manually add this statement.

Note: 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 190 shows the generated `BamkImpl` with `accounts` added.

Example 190: *BankImpl*

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

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

import java.lang.String;
import com.iona.schemas.wsaddressing.EndpointReferenceType;

public class BankImpl implements java.rmi.Remote
{
  Hashtable accounts = new Hashtable();

  public EndpointReferenceType get_account(String account_name)
  {
    return new
   com.iona.schemas.wsaddressing.EndpointReferenceType();
    }
}
```

get_account

The logical operation `get_account` is mapped to the `get_account()` method in the bank service's implementation object. `get_account()` does the following:

1. Checks the table of accounts to see if one with the given name already exists.
2. If one does exist, returns the endpoint reference to that account.
3. If no account with that name exists, it does the following:
   i. creates a new `AccountImpl` object
   ii. registers it as a transient servant with the bus.
   iii. returns an endpoint reference to the new account.

The `AccountImpl` object is registered as a transient servant because transient servants are guaranteed to have a unique `port` element in their in-memory contract and that the endpoint reference created for each

`AccountImpl` object will point to the correct servant. When using static servants, all endpoint 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 an endpoint reference for the new servant using `bus.createEndpointReference()`. This is the endpoint reference that is returned to the client. Using the returned endpoint reference, the client can create a service proxy to access the new `Account` instance.

Example 191 shows the fully implemented `get_account()`.

Example 191: *get_account()*

```
public EndpointReferenceType get_account(String account_name)
{
1   EndpointReferenceType ref =
      (EndpointReferenceType)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.createEndpointReference(name);

8    accounts.put(account_name, ref);
    }

8   return ref;
}
```

The code in Example 191 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 an endpoint reference for the newly registered transient servant.
8. Adds the new endpoint reference and account name to the table of accounts.
9. Returns the endpoint 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

Endpoint references provide all of the information needed to contact a remote service. They do not provide access to the contract defining the remote service or the interface used to create the interface. Therefore, your client application will need access to the following additional artifacts:

- the generated interface for the `Account` service. This interface will be generated into a file called `Account.java` by `wsdltojava`.
- a copy of the contract defining the `Account` service. This contract should be available from the endpoint.

Note: 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 service.
2. Invokes `get_account()` on the bank proxy.
3. Creates a service proxy for an account service using the returned endpoint reference.
4. Invokes operations in the account service 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 an account service. The version of `Bus.createClient()` used to create a service proxy from an endpoint reference takes two parameters:

- an endpoint reference
- the interface class for the referenced service

Example 192 shows the code for creating an account service proxy from an endpoint reference.

Example 192: *Creating an Account Service Proxy*

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

Code for the client main()

Example 193 shows the completed code for the bank client's main line.

Example 193: *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.wsaddressing.EndpointReferenceType;

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);
```

Example 193: *Code for Bank Client*

```
    String account_name;
    System.out.println("What is the name of the account?");
    System.in.read(account_name);

    EndpointReferenceType acctRef =
   bankProxy.get_account(account_name);

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

  // Invoke operations on acctProxy
    }
}
```

The code in Example 193 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 an endpoint reference for the desired account by invoking `get_account()` on the bank service proxy.
8. Uses the returned endpoint reference to create an account service proxy using `bus.createClient()`.

Using Endpoint References to Implement Callbacks

Overview

Another common use for endpoint references is to create callbacks from a service to a client. When creating a callback, the client creates a callback service to receive notices and registers it, using an endpoint reference, with the remote service. The remote service can then create a service proxy for the client's callback service 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 each have a callback service with two operations, `posting` and `done_posting`. The accounts receivable system would invoke `posting` to notify the client that it is not accepting new transactions. When it was done closing the books, the accounts receivable system 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 service implemented by the client. When using callbacks the client essentially plays a dual role. It implements a service, like a server process, and makes requests on a service.

Messages with references

The `Register` interface's `register_callback` operation sends an endpoint reference to a `Notify` service. The logical message definition for this interaction has one part, `ref`, that is of type `wsa:EndpointReferenceType` as shown in Example 194.

Example 194: *Message with a Reference*

```
<message name="regMessage">
  <part name="ref" type="wsa:EndpointReferenceType" />
</message>
```

The callback's interface

The interface for the callback service can be as complex or simple as your application requires. For this example, the callback service will only need two operations. One operation informs the client that the accounts receivable system is busy. The other operation informs the client that the accounts receivable service is ready to receive new posts. Neither operation requires 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 195 shows the `portType` element defining the callback service's interface.

Example 195: *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>
```

Accounts receivable system's interface

The account receivable system's interface needs one operation, `register_callback`, to register the client's callback service and create a proxy for it. In addition to the operation for registering the callback, the account receivable system's interface can have any number of logical operations to represent the other functionality it exposes. In this example, the accounts receivable system exposes three operations: `deposit`, `withdraw`, and `dailyPosting`. The client shown in this example only invokes `desposit` and `withdraw`. An administrative client would invoke `dailyPosting`.

Example 196 shows the `portType` element defining the accounts receivable system's interface.

Example 196: *Accounts Receivable 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 service's interface can be bound to any of the message formats supported by Artix. Because the account receivable system's interface includes an operation that has an endpoint 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's implementation object and the accounts receivable system's implementation object are registered as static servants, they can use any of the transports supported by Artix. In this example, HTTP is used.

Contract

Example 197 shows the complete contract used for the code generated in the following discussions about callbacks.

Example 197: *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:wsa="http://www.w3.org/2005/08/addressing"
  targetNamespace="http://www.iona.com/bus/demos/callbacks"
  name="BankService">
  <import location="http://www.w3.org/2005/08/addressing/ws-addr.xsd"
          namespace="http://www.w3.org/2005/08/addressing" />
  <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="wsa:EndpointReferenceType"/>
  </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 197: *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 197: *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 has two major parts to develop:

- The callback service'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 by servers. It will instantiate a servant for the callback service and register it with the bus.

Callback implementation

The callback service 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 that the status has been updated and it can stop waiting for the server.

Example 198 shows the code for the callback object.

Example 198: *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 198: *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 service.
3. Registers the callback service's servant with the bus so that it can receive requests.
4. Registers the callback service with the `Register` service.
5. Invokes operations on the `Register` service.
6. Checks the callback service to see if the `Register` service is posting.

Example 199 shows the code for client `main()`.

Example 199: *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.wsaddressing.EndpointReferenceType;

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 199: *Callback Client Main()*

```
5     bus.registerServant(servant, notifyName);

6     EndpointReferenceType ref =
    bus.createEndpointReference(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 199: *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 199 does the following:

1. Initializes a bus for the client.
2. Creates a proxy for the `Register` service.
3. Creates an instance of `NotifyImpl`.
4. Creates a servant to wrap the callback service.
5. Registers the servant with the bus.
6. Creates an endpoint reference for the callback service.
7. Registers the callback by invoking the `Register` service's `register_callback()` operation.
8. Ensures that the callback implementation cannot be modified by other threads before checking its state.
9. If the callback implementation's busy flag is set to 1, then the server is doing its daily posting and the client needs to wait.
10. Waits on the callback's implementation. 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 service 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.

Servant registration

In this example, the code that instantiates the `Servant` and registers it with the bus is standard Artix code. For more information see "Developing a Container Based Service" on page 64 or "Developing a Standalone Service" on page 75.

RegisterImpl

The accounts receivable system'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, only `register_callback()` and `dailyPosting()` are of interest. `register_callback()` is responsible for receiving the callback service's endpoint reference and instantiating a proxy for it. In this example, the proxy is stored in the objects `notify` member. `dailyPosting()` then invokes the callback service's operations to inform the client when the system is busy.

Example 200 shows the completed `RegisterImpl` class. The code in bold is added to the generated class by the user.

Example 200: *RegisterImpl*

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

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

import com.iona.schemas.wsaddressing.EndpointReferenceType;
import com.iona.jbus.*;
import java.lang.String;
```

Example 200: *RegisterImpl*

```
public class RegisterImpl implements java.rmi.Remote
{
  NotifyImpl notify;

  public void register_callback(com.iona.schemas.wsaddressing.EndpointReferenceType 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 service using the endpoint reference sent by the client.

dailyPosting()

`dailyPosting()` does the following:

1. Invokes the callback service's `posting` operation to notify the client that the system 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 system's `done_posting` operation to notify the client that the system is ready to handle new requests.

Migration Scenarios

Overview

With the release of Artix 4.0, Artix switched from using a proprietary reference format to using the WS-Addressing endpoint reference format. If you have existing applications that use the proprietary format, you should consider migrating those applications to the WS-Addressing standard.

Retaining proprietary references

Artix 4.0 retains support for the proprietary reference format and the associated APIs. This means the following:

- existing applications that use the proprietary format can simply be recompiled with out changing the code.
- new services written using Artix 4.0 can interoperate with older services if they are properly implemented. For more information see "Mixing references types" on page 323.

Migrating to WS-Addressing

Migrating your applications to use WS-Addressing endpoint references is a straight process. You would need to do the following:

1. Add a prefix definition to your contract for the WS-Addressing namespace.
2. Modify the `import` element in your contract to import the WS-Addressing schema instead of the Artix reference schema.
3. Replace the `reference:Reference` type with the `wsa:EndpointReferenceType` type.
4. Regenerate the stub and skeleton code for your applications.
5. Replace all instances of `Reference` with `EndpointReferenceType`.
6. Replace all instances of `createReference()` with `createEndpointReference()`.

Mixing references types

You can have applications that use both the proprietary references and the WS-Addressing endpoint references. However, they do not share the same wire format and are not interchangeable. If your application uses both styles, you must ensure that operations using proprietary references use the

old `Reference` type and the older APIs on both the client-side and service-side. If an operation expecting an Artix reference receives an endpoint reference it will throw an exception.

The same is true of operations that use endpoint references. They cannot process Artix references.

CHAPTER 13

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 two utility classes:

- `com.iona.jbus.utils.XMLUtils` provides an overloaded static method `fromXML()` for populating objects using XML data stored as a string. This class supports namespaces. It uses the JNI layer to parse the data which can make it inappropraite for large XML documents.
- `com.iona.jbus.util.StreamUtils` provides a static `fromXML()` method for converting XML data stored in an `InputStream` object as a Java object of the appropraite class. This method does not use the JNI layer and is very efficient. However, it does not support namespaces.

Note: `StreamUtils` is only available if you have installed patch 20080305 or higher.

Populating an object generated from an XML Schema type

If the object you are populating is generated to represent an XML Schema type, you can use the simple form of `fromXML()`. The signature for this form is shown in Example 201.

Example 201: *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 XML Schema 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 XML Schema document defining the data the object represents.

If, for example, your application works with student records whose structure is defined as an XML Schema 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 202.

Example 202: *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 202 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 XML Schema definition for the type a `WriteException` will be thrown.

Populating an object generated from an XML Schema element

If the object you are populating is generated to represent an XML Schema element, you can use the more flexible form of `fromXML()`. This form will work with both XML Schema types and XML Schema elements. The signature for this form is shown in Example 203.

Example 203: *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 XML Schema element from which the object was generated.
`QName typeName`	Specifies the QName of the XML Schema 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 XML Schema document defining the data the object represents.

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

If we changed Example 202 so that `studentRec` was defined as an XML Schema element instead of a complex type, the code for populating the object would be similar to that shown in Example 204.

Example 204: *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");
```

Example 204: *Populating an Object from XML*

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

The code in Example 204 differs from the code in Example 202 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 XML Schema definition for the element a `WriteException` will be thrown.

Populating an object from an XML document stored as a Stream

If the XML data from which you want to create an object from is stored in an `InputStream` object you can use the stream-based `com.iona.jbus.util.StreamUtils.fromXML()` method. This method will work with both XML Schema types and XML Schema elements. The signature for this form is shown in Example 205.

Example 205: *Stream form of fromXML()*

```
static Object fromXML(Class cls, String wsdlPath, InputStream is)
throws ReadException;
```

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

`Class cls`	Specifies the `Class` object of the class into which the XML is being converted.
`String wsdlPath`	Specifies the path to the WSDL document or XML Schema document containing the type definition for the XML data.
`InputStream is`	Specifies the `InputStream` object containing the XML data.

If we changed Example 202 so that `studentRec` was stored in a `FileInputStream`, the code for populating the object would be similar to that shown in Example 206.

Example 206: *Populating an Object from XML Using Streams*

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

StudentRec student =
  (StudentRec)StreamUtils.fromXML(StudentRec.class,
                                  "./grader.wsdl",
                                  file);
```

If the XML data passed into `fromXML()` does not conform to the XML Schema definition for the element a `ReadException` 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 two utility classes:

- `com.iona.jbus.utils.XMLUtils` provides the overloaded static method `toXML()` for converting objects into their XML form and storing the results as a string. These methods support XML namespaces. However, they use the Artix JNI layer and incurs heavy processing penalties when working with large chunks of data.
- `com.iona.jbus.util.StreamUtils` provides a static `toXML()` method for converting objects into thier XML form and storing the results in an `OutputStream` object. This method does not use the JNI layer and is very efficient. However, it does not support namespaces.

  Note: `StreamUtils` is only available if you have installed patch 20080305 or higher.

Artix objects that represent an XML Schema type

If the object you are converting into XML was generated by Artix to represent an XML Schema type you can use the simplest form of `toXML()`. The signature for this form is shown in Example 207.

Example 207: *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 XML Schema document defining the data the object represents.

Objects that represent an XML Schema type

If you have an object, that was not generated by Artix, that represents an XML Schema type and you have access to the XML Schema 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 XML Schema type the object represents. The signature for this form is shown in Example 208.

Example 208: *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 XML Schema 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 XML Schema document defining the data the object represents.

Objects that represent an XML Schema element

If you have an object, that represents an XML Schema element and you have access to the XML Schema 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 XML Schema element the object

represents. It also allows you to convert an object that represents an XML Schema type by specifying the type's QName. The signature for this form is shown in Example 209.

Example 209: *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 XML Schema element represented by the object.
`QName typeName`	Specifies the QName of the XML Schema 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 XML Schema document defining the data the object represents.

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

Storing the results in a stream

If you have an object representing an XML object and you want to convert it into an `OutputStream` object containing the XML, you would use the stream-based `com.iona.jbus.util.StreamUtils.toXML()` method. The signature for this method is shown in Example 210.

Example 210: *Stream-based toXML()*

```
static void toXML(Object obj, String wsdlPath, OutputStream os)
throws WriteException;
```

It takes three 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 wsdlPath`	Specifies the path to the contract or XML Schema document defining the data the object represents.
`OutputStream os`	Specifies the `OutputStream` into which the XML is written. This can be any implementation of the `OutputStream` interface.

Converting References into XML

Overview

Artix references are defined with in an Artix specific XML Schema 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 14

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. `MessageContext` objects, or *message contexts*, are primarily used in writing handlers, but can also be used to store metadata about messages or to pass state information into or out of the message handling chain. Generally, this metadata is not passed across the wire with the message.

Artix extends the JAX-RPC `MessageContext` interface to create Artix `IONAMessageContext` objects, or *Artix message contexts*. The Artix message contexts provide a consistent, thread safe mechanism for passing supplemental information along with request and reply messages. This supplemental information can include SOAP headers, GIOP context objects, transport attributes, and MIME type definitions.

Artix message context hierarchy

All message contexts in Artix are based on the JAX-RPC `MessageContext` interface as shown in Figure 5.

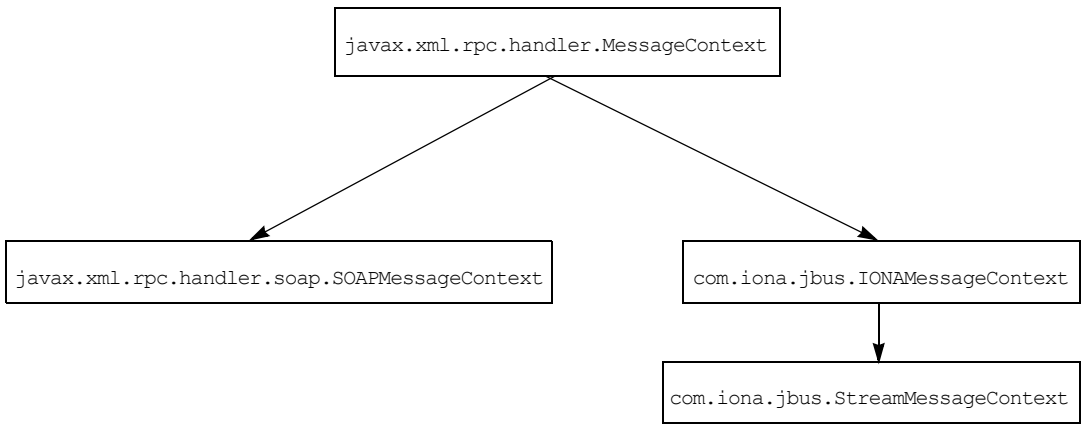

Figure 5: *Artix Message Context Hierarchy*

All of the Artix operations that return a message context return an object that implements the `MessageContext` interface. Depending on where you are in your code and what properties you want to access, you can cast the returned message context into an object that implements one of the other interfaces.

How Artix uses message contexts

In Artix, message contexts are thread-specific objects that are managed by an instance of the Artix bus. Each bus instance creates a *context registry* to manage its message contexts. The context registry manages the list of registered context properties and one `MessageContext` instance for each thread the bus has spawned. This mechanism ensures that message contexts remain tied to the messages for which they are created.

Artix consumers, however, do have a number of Artix specific context properties that survive beyond the life of a single message. These properties contain information used to configure the bindings and transports used by the consumer. The values of these properties persist until they are reset by application code. For more information see "Working with Transport Attributes" on page 387.

Artix extensions to message contexts

Artix extends the JAX-RPC `MessageContext` interface to create an Artix-specific `IONAMessageContext` interface. This interface is used to implement Artix message contexts that hold information which is to be written out on the wire or used to alter how messages are sent and received. To ensure that these properties remain attached to the correct message in the sequence, Artix message contexts use two containers:

- a request context container that hold properties associated with messages that travel from a consumer to a service
- a reply context container that holds properties associated with messages that travel from a service to a consumer

This is shown in Figure 6.

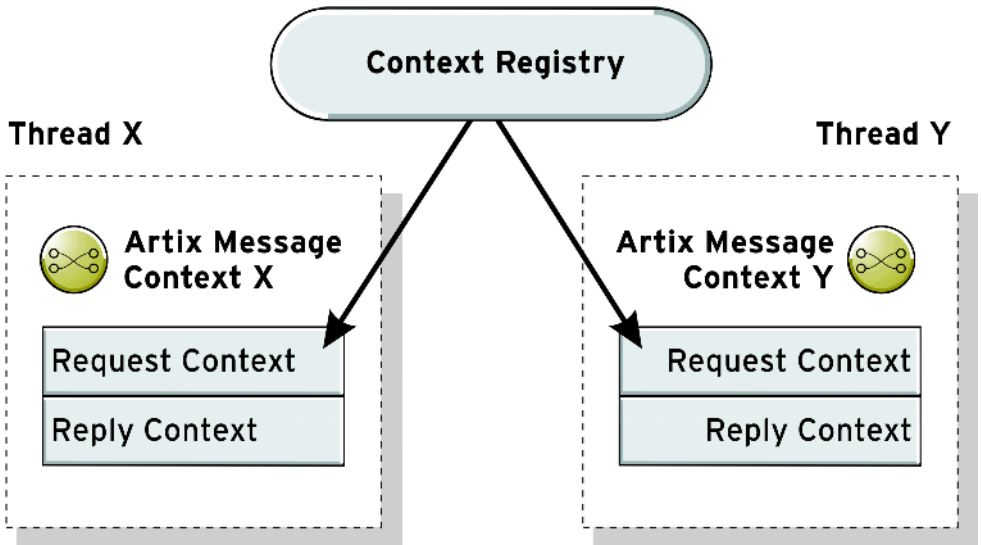

Figure 6: *Overview of the Message Context Mechanism*

The request and reply context containers hold separate instances of each property. So, a property can have one value for requests and one for replies. Some properties are specific to a particular container. For example, the HTTP properties are different for requests and replies.

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 264.
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. The JAX-RPC interface allows you to set properties in the message context as name value pairs. These properties can then be accessed as a message passes along the messaging chain. For more information see "Working with JAX-RPC MessageContext Objects" on page 347.

The Artix interface allows you to manipulate properties that are used to create message headers or to change transport attributes. In addition, the Artix interface, because it inherits from the JAX-RPC interface, can also access any property set using the JAX-RPC message contexts. For more information see "Working with IonaMessageContext Objects" on page 353.

The SOAP interface, 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 575.

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 them out when requested by the application.

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 211 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 211: *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 methods shown in "Getting a Bus" on page 98.
2. Call `getContextRegistry()` on the returned bus to get a reference to the context registry.

Example

Example 212 shows an example of getting the context registry from within the implementation object of an Artix service.

Example 212: *Getting the Context Registry*

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

public class Atherny
{
// get the bus
```

Example 212: *Getting the Context Registry*

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

    ...
    }
```

The code in Example 212 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 MessageContext Object for a Thread

Overview

To ensure thread safety, the context registry creates a message context 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` you can access the Artix specific context properties.

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 213 shows the signature for `getCurrent()`.

Example 213: *getCurrent()*

```
javax.xml.rpc.handler.MessageCcontext ContextRegistry.getCurrent();
```

Example 214 shows how to get an message context from the context registry.

Example 214: *Getting a Message Context*

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

Example 214: *Getting a Message Context*

```
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 214 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 servant-specific 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 215 shows the signature for `getCurrentMessageContext()`.

Example 215: *getCurrentMessageContext()*

```
javax.xml.rpc.handler.MessageCcontext getCurrentMessageContext();
```

Example 216 shows how to get an message context using the `DispatchLocals` interface.

Example 216: *Getting a Message Context*

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

Example 216: *Getting a Message Context*

```
import com.iona.jbus.*;

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

Working with JAX-RPC MessageContext Objects

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 the components along the messaging chain. 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 messaging 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 15 lists the context properties used by Artix.

Table 15: *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 570.

Table 15: *Artix Context Properties*

Property	Description
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 570.
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 570.
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 570.
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 570.
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 570.
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 570.
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 570.
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 570.

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 217 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 217: *MessageContext.setProperty()*

```
void setProperty(Sting name, Object value);
```

The scope of the property depends on where in the messaging chain the property is set into the context. Properties set at the implementation level 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 539.

Example 218 shows the code for setting a property in the request context.

Example 218: *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 218 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 219 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 219: *MessageContext.getProperty()*

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

Example 220 shows the code for getting a property from the request context.

Example 220: *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 220 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 221 shows the signature for `removeProperty()`. It takes a single parameter, `name`, that represents the name of the property you wish to remove.

Example 221: *MessageContext.removeProperty()*

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

Example 222 shows the code for removing a property from the message context.

Example 222: *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 222 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 223 shows the code for seeing if a property is set in the message context.

Example 223: *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 IonaMessageContext Objects

Overview

Artix extends the `MessageContext` interface defined by JAX-RPC to support some of Artix's more advanced features. This extended interface is the `IonaMessageContext` interface. Message contexts that are accessed using this interface are referred to as Artix message contexts. They are used to store complex data types that are used for adding header elements to messages or to programatically define certain binding and transport details.

In this section

This section discusses the following topics:

How Properties are Stored in Artix Message Contexts

Overview

Artix message contexts store data that must be used by both the Java components of Artix and the C++ components of Artix. In addition, the properties for a transport may differ depending on the direction a message is travelling and a property may or may not be populated depending on where in the request/reply sequence you access it. Transport properties and message headers must also be preserved across multiple Artix endpoints. For example, the HTTP properties that are available for a client to set on a request are different from the HTTP properties that it can access for a reply. Also, the HTTP properties for a reply are only available to the client after a reply is received. If the invocation chain involves a router, the router must also preserves both the request's HTTP properties and the response's HTTP properties.

To make these capabilities possible Artix message contexts use two additional components:

- The *context registry* contains the list of all possible properties that can be stored in the Artix message contexts for the current bus.
- Two *context containers* store individual instances of each Artix message context.

Property registration

Properties stored in an Artix message context are defined as XML Schema complex types. Each XML Schema complex type represents one Java object that conforms to the mappings described in "Using XML Schema Complex Types" on page 148. Before the property can be placed into an Artix message context it must be registered with the context registry using the context registry's `registerContext()` method. For more details on registering properties with the context registry see "Registering Context Types" on page 369.

Each of the transports shipped with Artix has a set of properties that are managed using Artix message contexts. The transports automatically register all of the properties they use when the transport is loaded. For more information see "Working with Transport Attributes" on page 387.

Context containers

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 7 shows how context properties are passed with messages.

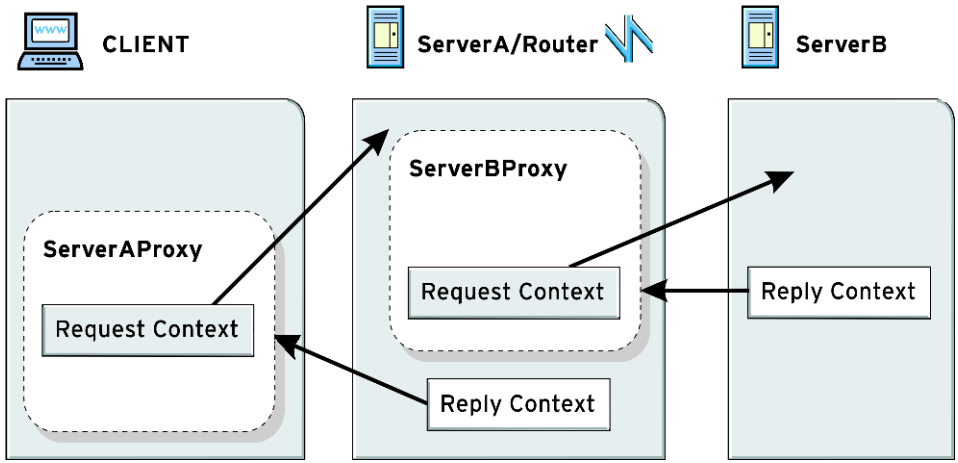

Figure 7: *Contexts Passed Along Request/Reply Chain*

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 into an Artix Message Context

Overview

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 property is set by the sender of the message
- the property is set using the context's setter methods

Received properties

When Artix receives a message the transport layer will populate the appropriate properties in the Artix message context. The SOAP and CORBA bindings will populate the appropriate properties if headers are attached to messages. In addition, other Artix plug-ins that have access to a message can also set properties based on the content of a received 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.

Artix message context setter methods

Artix message contexts have two setter methods: `setReplyContext()` and `setRequestContext()`. Example 224 shows the signature for these methods.

Example 224: *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`.

Procedure for setting a property

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

Example 225 shows the code for setting a property in the request context.

Example 225: *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 225 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.

Working with Properties from an Artix Message Context

Overview

Once a property is set in an Artix message context you can retrieve the property and manipulate its contents. Properties in an Artix message context are Java objects, so you manipulate them as you would any other Java 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 226 shows the signature for these methods.

Example 226: *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.

Manipulating 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 227 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.

Example

Example 227 shows the code for getting a property from the request context.

Example 227: *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 227 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`.

Special Artix Properties

Overview

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.

Oneway property

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 228 shows code for checking if a request is oneway.

Example 228: *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 228 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.

Correlation ID property

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 Message Headers

Artix message contexts are used to add headers to messages that are sent using payload formats that support message headers.

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 for sending metadata such as security tokens or session information that is tangential to the logic involved in processing the request.

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 used by the application.

Note: If you change the payload format used by the application, your code will continue to work. However, the header information stored in the context will not be transmitted.

Procedure

To send customer header information in a context you need to do the following:

1. Define an XML Schema 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 264.
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 chapter

This chapter discusses the following topics:

Defining Context Data Types

Overview

Contexts can store data of any XML Schema type that is derived from `xsd:anyType`. In other words, a context data type can be any primitive, simple, or complex XML Schema type.

When creating a context whose type is an XML Schema primitive type or a native XML Schema 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 XML Schema 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 typically appropriate to define a context data type (or types) in a separate schema file, rather than including the definition in the application's contract. This approach is logical because contexts are normally used to implement features independent of any particular service.

To define a complex context data type, *`ContextDataType`*, in the namespace, *`ContextDataURI`*, you define a context schema following the outline shown in Example 229.

Example 229: *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, `message`, contains a message string. The data type for this header, `SOAPHeaderInfo`, is shown in Example 230.

Example 230: *Header Context Data Definition*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://schemas.iona.com/types/context"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:complexType name="SOAPHeaderInfo">
    <xs:sequence>
      <xs:element name="originator" type="xs:string"/>
      <xs:element name="message" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs: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 wishes to register and use a context of the defined type.

For more information on type factories see "Working with Artix Type Factories" on page 261.

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()` requires the `QName` for the context and the `QName` of the data type stored in the context.

Registering a context adds a type factory reference to the context registry's 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 for Use as a SOAP Header

Overview

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

Example 231: *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`.

Example

For example, to register a SOAP header property of the type defined in Example 230 on page 368 you would use code similar to Example 232.

Example 232: *Registering a SOAP Header Property*

```
SOAPHeaderInfoTypeFactory fact = new
   SOAPHeaderInfoTypeFactory();
// Bus, bus, obtained earlier
bus.registerTypeFactory(fact);
```

Example 232: *Registering a SOAP Header Property*

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

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 232 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 for Use as a CORBA Header

Overview

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

Example 233: *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.

Example

For example, to register a CORBA header property of the type defined in Example 230 on page 368 you would use code similar to Example 234.

Example 234: *Registering a Property as a CORBA Header*

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

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");
```

Example 234: *Registering a Property as a CORBA Header*

```
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 234 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.

In this section

This section discusses the following topics:

The Contract

Overview

The contract used for this example imports the XSD file, `SOAPcontext.xsd`, that defines the SOAP header property's data type in Example 230 on page 368. 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

Example 235 on page 375 shows the contract used to define the service used in this example.

Example 235: *SOAP Header WSDL*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorld" targetNamespace="http://www.iona.com/custom_soap_interceptor"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/custom_soap_interceptor"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <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="sayHiRequest"/>
  <message name="sayHiResponse">
    <part element="tns:responseType" name="theResponse"/>
  </message>
  <message name="greetMeRequest">
    <part element="requestType" name="me"/>
  </message>
  <message name="greetMeResponse">
    <part element="responseType" name="theResponse"/>
  </message>
```

Example 235: *SOAP Header WSDL*

```
  <portType name="Greeter">
    <operation name="sayHi">
      <input message="tns:sayHiRequest" name="sayHiRequest"/>
      <output message="tns:sayHiResponse" name="sayHiResponse"/>
    </operation>
    <operation name="greetMe">
      <input message="greetMeRequest" name="greetMeRequest"/>
      <output message="greetMeResponse" name="greetMeResponse"/>
    </operation>
  </portType>
  <binding name="GreeterSOAPBinding" type="Greeter">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHi">
      <soap:operation soapAction="" style="document"/>
      <input name="sayHiRequest">
        <soap:body use="literal"/>
      </input>
      <output name="sayHiResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="greetMe">
       <soap:operation soapAction="" style="document"/>
       <input name="greetMeRequest">
         <soap:body use="literal"/>
       </input>
       <output name="greetMeResponse">
         <soap:body use="literal"/>
       </output>
    </operation>
  </binding>
  <service name="SOAPService">
    <port binding="Greeter_SOAPBinding" name="SoapPort">
      <address location="http://localhost:9000"/>
    </port>
  </service>
</definitions>
```

Generating the Classes for the Header

Overview

In order to generate the proper classes to support the custom SOAP header you need to run the `wsdltojava` tool on the SOAP headers schema file separately. This will create the classes needed to work with the header. You will need to add these classes to your application.

Procedure

To generate the code for the header save header's schema file to a file called `SOAPcontext.xsd`. Then run the following command:

```
wsdltojava SOAPcontext.xsd
```

The file *InstallDir*`/artix_5.1/cxx_java/samples/advanced/custom_soap_header/etc/contextData.xsd` also defines `SOAPHeaderInfo`.

The Client

Overview

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

Example 236 on page 378 shows the code for the client.

Example 236: *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);
```

Example 236: *Client Code*

```
2     QName name =
        new QName("http://www.iona.com/custom_soap_interceptor",
                  "SOAPService");
      QName portName = new QName("","SoapPort");

      String wsdlPath = "../../etc/hello_world.wsdl";

     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);
      Soap impl = (Soap)service.getPort(portName, Soap.class);

3     SOAPHeaderInfoTypeFactory fact =
        new SOAPHeaderInfoTypeFactory();
      bus.registerTypeFactory(fact);

4     ContextRegestry contReg = bus.getContextRegistry();

5     QName name = new QName("", "SOAPHeaderInfo");

6     QName type =
        new QName("http://schemas.iona.com/types/context",
                  "SOAPHeaderInfo");

7     QName message =
        new QName("http://schemas.iona.com/custom_header",
                  "header_content", "soap_header");

8     contReg.registerContext(name, type, message, "header_info");

9     SOAPHeaderInfo header = new SOAPHeaderInfo();
      header.setOriginator("Client");
      header.setMessage("Hello from Client.!");
```

Example 236: *Client Code*

```
10      IonaMessageContext context =
          (IonaMessageContext)contReg.getCurrent();

11      context.setRequextContext(name, header);

12      String string_out;

        string_out = impl.sayHi();
        System.out.println(string_out);
13      string_out = impl.greetMe("Chris");
        System.out.println(string_out);

14      SOAPHeaderInfo replyInfo =
       (SOAPHeaderInfo)(context.getReplyContext(name));
        System.out.println("Context from Server: " + replyInfo);
        bus.shutdown(true);
        }
}
```

The code in Example 236 on page 378 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.
8. Registers the property with the context registry.
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 `sayHi()`. The SOAP header property is placed into the SOAP header of the request and sent to the server.

13. Invokes `greetMe()`. The SOAP header property is placed into the SOAP header of the request and sent to the server.
14. Retrieves the SOAP header that is returned from the service and displays its contents.

The Service

Overview

A service that works with SOAP headers need to do three things:

1. Register the type factory for the SOAP header's data type.
2. Register the SOAP headers property with the context registry.
3. Extract the SOAP header from the request.

Registering the property in the service

The service 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 service's `busInit()`.

Example 237 on page 382 shows the code for the service's `busInit()`.

Example 237: *Registering a Context in busInit()*

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

import java.net.*;
import java.io.*;

public class HelloWorldPlugIn extends BusPlugIn
{
 private HelloWorldImpl service;

  public void busInit() throws BusException
  {
1   Bus bus = getBus();

2   QName name =
      new QName("http://www.iona.com/custom_soap_interceptor",
                "SOAPService");
    Servant servant =
      new SingleInstanceServant(new SoapImpl(),
                                "../../etc/hello_world.wsdl", bus);
    bus.registerServant(servant, name, "SoapPort");
```

Example 237: *Registering a Context in busInit()*

```
3     SOAPHeaderInfoTypeFactory fact =
        new SOAPHeaderInfoTypeFactory();
      bus.registerTypeFactory(fact);

4     ContextRegestry contReg = bus.getContextRegistry();

5     QName propName =  new QName("", "SOAPHeaderInfo");

6     QName propType =
                new QName("http://schemas.iona.com/types/context",
                          "SOAPHeaderInfo");

7     QName message =
        new QName("http://schemas.iona.com/custom_header",
                  "header_content", "soap_header");

8     contReg.registeContext(propName, propType,
                             message, "header_info");

    }
    ...
}
```

The code in Example 237 on page 382 does the following:

1. Gets 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.
7. Builds the `QName` for the message defining the SOAP header.
8. Registers the property with the context registry.

Extracting the SOAP header

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 238 shows the code for the `GreeterImpl` implementation object.

Example 238: *Implementation of the Greeter Service*

```
import java.net.*;
import java.rmi.*;
import javax.xml.namespace.QName;

import com.iona.jbus.*

public class SoapImpl
{
  public String greetMe(String stringParam)
  {
1    IonaMessageContext context =
   (IonaMessageContext)DispatchLocals.getCurrentMessageContext();

4   QName name =  new QName("", "SOAPHeaderInfo");

5   SOAPHeaderInfo header =
     (SOAPHederInfo)context.getRequestContext(name);

6   System.out.println("SOAP Header Originator:
                       "+header.getOriginator());
    System.out.println("SOAP Header message:
                       "+header.getMessage());

7   SOAPHeaderInfo replyInfo = new SOAPHeaderInfo();
    replyInfo.setOrginator("Servier");
    replyInfo.setMessage("Success! :)");

8   context.setReplyContext(name,replyInfo);
```

Example 238: *Implementation of the Greeter Service*

```
9     return "Hello Artix User: "+stringParam;
      }
}
```

The code in Example 238 on page 384 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. Instantiates a new SOAP header to send back to the client.
8. Sets the new SOAP header into the reply context so it can be returned to the client.
9. Returns the results of the operation to the client.

CHAPTER 16

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:

Setting i18n Attributes page 463

How Artix Stores Transport Attributes

Overview

Artix uses the context mechanism described in "Using Message Contexts" on page 337 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 Bindings and Transports, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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 342.
2. Get the message context for the current application as shown in "Getting the MessageContext Object for a Thread" on page 344.
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 IonaMessageContext Objects" on page 353. 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 16.

Table 16: *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 16: *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
FTP Connection Policy	FTP_CONNECTION_POLICY
FTP Client Naming Policy	FTP_CLIENT_NAMING_POLICY
FTP Server Naming Policy	FTP_SERVER_NAMING_POLICY
FTP Connection Credentials	FTP_CREDENTIALS
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 17 lists the data types for each of the configuration contexts.

Table 17: *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

Table 17: *Configuration Context Classes*

Context	Class
JMS Server Header Attributes	`com.iona.schemas.transports.jms.context.JMSServerHeadersType`
FTP Connection Policy	`com.iona.schemas.transports.ftp.context.ConnectionPolicyType`
FTP Client Naming Policy	`com.iona.schemas.transports.ftp.context.ClientNamingPolicyType`
FTP Server Naming Policy	`com.iona.schemas.transports.ftp.context.ServerNamingPolicyType`
FTP Connection Credentials	`com.iona.schemas.transports.ftp.context.CredentialsType`
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`

Getting IP Attributes

Overview

Artix provides a context that enables you to access data from the IP socket layer. Currently, the only supported IP attribute is the client IP address, which is accessible through the *client address context*.

Client address context

The client address context is a server-side request context that contains the IP address (or hostname) of the requesting client. This context can be useful if you want a simple way of identifying clients—for example, for the purposes of logging requests on the server side.

WARNING: The client address context is *not* a secure way to identify clients. If you need to be certain of the client's identity, use one of the authentication techniques described in the *Artix Security Guide*.

Enabling the client address context

To enable the client address context on the server side, insert the following setting into the relevant scope of your server's `.cfg` configuration file:

```
# Artix Configuration File
plugins:bus:register_client_context = "true";
```

This setting causes the Bus to read the client's IP address from the IP socket layer each time the server receives a message from a client. The IP address is then inserted into a client address context, which is accessible to the server application code.

Note: The default setting is `false`, thus disabling the client address context. This is to avoid any unnecessary performance overhead when this feature is not needed.

Getting the client address on the server side

The context containing the client's IP address, `CLIENT_ADDRESS_CONTEXT`, is available in the server's request context container, *after* a request from the client is received by the transport layer. To access the client's IP address on the server side, use the code fragment shown in Example 239.

Example 239: *Getting the Client's Address*

```
private void printClientAddress()
{
try
{
  IonaMessageContext contextImpl = (IonaMessageContext)DispatchLocals.getCurrentMessageContext();
  String clientAddress =
    (String)contextImpl.getRequestContext(ContextConstants.CLIENT_ADDRESS_CONTEXT);
  System.out.println("client address: " + clientAddress);
}
catch (ContextException ctex) {
ctex.printStackTrace();
}
```

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 the following 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.
- FTP Transport Settings - specifies the attributes to use when establishing an FTP connection for the FTP transport.

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 240. 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 240: *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 241, 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 241: *getContext()*

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

You can also use the `setContext()` method, shown in Example 242, 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 242: *setContext()*

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

Setting HTTP Attributes

Overview

Artix uses five 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 fifth context is used to store custom HTTP header properties. It is of type `com.iona.schemas.transports.header.CustomHeaders`.

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_CONTEXTS`, 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 243 shows the code for setting the `AutoRedirect` property for a client.

Example 243: *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_CONTEXTS, true);

5  httpAtribs.setAutoRedirect(true);

   // make proxy invocations
```

The code in Example 243 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 18 shows the attributes that are valid in the outgoing HTTP client context.

Table 18: *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 18: *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 18: *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 18: *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 18: *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 18: *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 18: *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_CONTEXTS`, 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 244.

Example 244: *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_CONTEXTS, true);

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

The code in Example 244 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 19 shows the attributes that are valid in the incoming HTTP client context.

Table 19: *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 19: *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_CONTEXTS`, 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 245 shows the code for setting the `RedirectURL` attribute for a server.

Example 245: *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_CONTEXTS, true);

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

The code in Example 245 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 20 shows the attributes that are valid in the outgoing HTTP server context.

Table 20: *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 20: *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 20: *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 20: *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 20: *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_CONTEXTS`, 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 246.

Example 246: *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_CONTEXTS, true);

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

The code in Example 246 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 19 shows the attributes that are valid in the incoming HTTP server context.

Table 21: *Incoming HTTP Server Attributes*

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

Table 21: *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 21: *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 21: *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 398.

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.

Side effects

A side affect of setting the server's endpoint URL using contexts is that the following configuration variables are ignored:

- `policies:soap:server_address_mode_policy:publish_hostname`
- `policies:at_http:server_address_mode_policy:publish_hostname`

The endpoint addresses advertised by the WSDL publish service will reflect the values set in the configuration context, not the values set in the configuration file.

Example

Example 247 shows how to set the HTTP Endpoint URL programatically.

Example 247: *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);
```

Example 247: *Setting the HTTP Endpoint URL*

```
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 247 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.

Adding Custom HTTP Header Properties

Overview

The HTTP header can be used to store a variety of properties that are not a part of HTTP specification. Commonly people will store the SOAP action or basic authentication credentials in the HTTP header. You could also place properties in the HTTP head that are used by a RESTful service.

The context used to store custom HTTP header properties are stored in the Artix message context. It is valid in both the request container and the reply container. Both providers and consumers can set and inspect the custom HTTP header properties.

Getting the context

You access the custom HTTP header context using the standard get method for the desired context container. As shown in Example 248, the key used to access the context is `ContextConstants.TRANSPORT_CUSTOM_HEADERS`.

Example 248: *Accessing the Custom HTTP Properties Context*

```
import com.iona.schemas.transports.header.CustomHeaders.

ContextRegistry registry = bus.getContextRegistry();
IonaMessageContext context = (IonaMessageContext)registry.getCurrent();

CustomHeaders headers =
  (CustomHeaders)context.getRequestContext(ContextConstants.TRANSPORT_CUSTOM_HEADERS);
```

The returned context should be cast into a `com.iona.schemas.transports.header.CustomHeaders` object.

How properties are stored in the context

The `CustomHeaders` object stores properties as an array of `com.iona.schemas.transports.header.CustomHeader` objects. `CustomHeader` objects have two fields:

- `Name` stores the name of the custom property.
- `Value` stores the value of the costom property.

Setting properties into the context

To add a custom property to the HTTP header do the following:

1. Get the customer header context from the desired context container as shown in "Getting the context".
2. Create an array of `CustomHeader` objects and populate their fields.
3. Add the `array of CustomHeader` objects to the context using the `CustomHeaders.setCustom_headers()` method.

   `setCustom_headers()` takes an array of `CustomHeader` objects and sets the array as the set of custom properties to add to the HTTP header. Any existing properties will be overwritten. To add properties to an existing set of properties see "Manipulating the property set".

Example 249 shows an example of setting custom properties into the HTTP header.

Example 249: *Setting a Custom Property Set into the Context*

```
CustomHeader header = new CustomHeader();
header.setName("MyHeader");
header.setValue("\"MyValue\"");

ArrayList props = new ArrayList();
props.add(header);

if (props.size() != 0)
  {
    headers.setCustom_headers((CustomHeader[])props.toArray(new CustomHeader[props.size()]));
}
```

Manipulating the property set

Once you have access to the custom header context, you can use that to get access to the set of custom properties stored in the context using the `CustomHeaders.getCustom_headers()` method. As shown in Example 250, `getCustom_headers()` returns an array of `CustomHeader` objects.

Example 250: *Getting the Set of Custom Properties*

```
CustomHeaders reply_headers =
  (CustomHeaders)context.getReplyContext(ContextConstants.TRANSPORT_CUSTOM_HEADERS);
CustomHeader[] headers = reply_headers.getCustom_headers();
```

Any manipulation of the returned array is reflected in the contents of the custom header context. Therefore, if you wish to add a new property to an existing set of properties, you can add it to the returned array.

Note: Changing the contents of a consumer's reply context or a server's request context has no lasting effect because they are wiped out for each invocation.

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 251 shows the code for getting the CORBA principle in a server.

Example 251: *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 251 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 252 shows the code for setting the CORBA principle for a client request.

Example 252: *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 251 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 22 describes the attributes stored in the MQ connection attributes context.

Table 22: *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 22: *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 431.

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 253 shows code for specifying the queue and queue manager to use when making a request.

Example 253: *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 253: *Setting the Client's QueueManager and QueueName*

```
5  connect.setQueueManager("Bloggy");
6  connect.setQueueName("TalkBack");

7  // Make invocation on proxy
```

The code in Example 253 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 23.

Table 23: *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 23: *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 254 shows code for setting a client's connection to use XA style transactionality for a request.

Example 254: *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 253 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 135.

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 24 describes the attributes stored in the MQ message attributes context.

Table 24: *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 24: *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 436.

Table 24: *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 437.
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 438.
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 440.
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 25.

Table 25: *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 255 shows code for setting a request message descriptor's CorrelationStyle message Id.

Example 255: *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 255 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 135.

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 26.

Table 26: *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 256 shows code for setting a request message descriptor's `Persistence` field to `MQPER_PERSISTENT`.

Example 256: *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 256: *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 256 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 135.

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 27.

Table 27: *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 27: *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 257 shows code for setting a request message descriptor's Format field to MQPER_STRING.

Example 257: *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 257 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 135.

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 28.

Table 28: *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 258 shows code for setting a request message descriptor's `Report` field to `MQRO_DISCARD_MSG`.

Example 258: *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 258: *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 258 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 135.

Setting JMS Attributes

Overview

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 29 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 29: *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 29: *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 259 shows how to create a set of user-defined properties and set them on a client request's JMS message.

Example 259: *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 259 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 260 shows the methods for accessing the `TimeOut` value on a `JMSClientHeadersType` object.

Example 260: *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 444. 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 261 shows the code for setting the JMS client attributes for a request.

Example 261: *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 262 shows code for checking the `JMSCorrelationID` of a response.

Example 262: *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 444. 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 263 shows the code for setting the JMS header attributes for a response.

Example 263: *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 263: *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 262 shows code for checking a request's `JMSRedilvered` flag.

Example 264: *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 398.

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 265 shows how to set the JMS broker connection properties on an Artix JMS client.

Example 265: *Setting the JMS Connection Info*

```
1  ContextRegistry registry = bus.getContextRegistry();

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

Example 265: *Setting the JMS Connection Info*

```
3  ContextContainer cnt = registry.getConfigurationContext(name,
                                                           "SoapPort",
                                                             true);

4  JMSConnectionSecurityInfoType info =
     (JMSConnectionSecurityInfoType)cnt.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 265 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. Creates the service proxy.

Setting FTP Attributes

Overview

The attributes used to configure an FTP connection are split into four contexts:

- one for setting the policies used to connect to the FTP daemon.
- one for setting the credentials to use when connecting to the FTP daemon.
- one for setting the naming scheme implementation to use for Artix clients.
- one for setting the naming scheme implementation to use for Artix servers.

These settings are all controlled through 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 398.

Artix clients can dynamically set the scan interval used by the FTP transport. and can dynamically adjust the length of time they will wait for a response before timing out.

In this section

This section discusses the following topics:

Setting FTP Connection Policies

Overview

When setting the FTP connection policies you access them using the `FTP_CONNECTION_POLICY` tag. The FTP connection policy context information is returned as a `ConnectionPolicyType` object. All of the connection policies are valid when set in the configuration context. In addition, Artix clients can set the scan interval policy and the receive timeout policy in their request contexts.

Setting the connection mode

The FTP connection mode is set using a `com.iona.schemas.transports.ftp.context.ConnectionModeType` object. `ConnectionModeType` is an enumeration whose values are described in Table 30.

Table 30: *ConnectionMode Values*

Value	Artix API for Setting	Description
`active`	`setConnectMode(ConnectModeType.fromString("active"))`	Specifies that Artix controls the connection to the FTPD.
`passive`	`setConnectMode(ConnectModeType.fromString("passive"))`	Specifies that the FTPD controls the connection.

Example 266 shows code for setting the connection mode to `passive`.

Example 266: *Setting the FTP Connection Mode*

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

2  ContextRegistry contReg = bus.getContextRegistry();

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

Example 266: *Setting the FTP Connection Mode*

```
4  ContextContainer ctn = registry.getConfigurationContext(name,
                                                           "SoapPort",
                                                             true);

5  ConnectionPolicyType policy =
       (ConnectionPolicyType)ctn.getContext(ContextConstants.FTP_CON
       NECTION_POLICY, true);

6  policy.setConnectionMode(ConnectionModeType.fromString("passive"
       ));

7  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 266 does the following:

1. Imports the package containing the FTP connection policy attributes context type.
2. Gets the context registry.
3. Creates the service's QName.
4. Gets the Artix configuration context from the context registry.
5. Gets the FTP connection attributes context from the context.
6. Sets the connection mode.
7. Creates the proxy.

For more information about working with Artix enumerated types, see "Using Enumerations" on page 135.

Setting the connection timeout

The FTP connection time out determines the number of milliseconds Artix will spend in attempting to connect to the FTPD before timing out. It is set using `setConnectTimeout()`. The value is specified as an integer as shown in Example 267.

Example 267: *Setting the Connection Timeout Policy*

```
ConnectionPolicyType policy =
    (ConnectionPolicyType)ctn.getContext(ContextConstants.FTP_CON
    NECTION_POLICY, true);

policy.setConnectTimeout(10000);
```

Setting the scan interval

The scan interval determines the number of seconds that Artix waits before rescaning the remote message repository for new messages. In addition to being settable in the configuration context, the scan interval can also be set by Artix clients using the request context.

It is set using `setScanInterval()`. The value is specified as an integer as shown in Example 268.

Example 268: *Setting the Scan Interval in a Client*

```
IONAMessageContext cont = (IONAMessageContext)
  DispatchLocals.getCurrentMessageContext();

ConntectionPolicyType policy = (ConnectionPolicyType)
  cont.getRequstContext(ContextConstants.FTP_CONNECTION_POLICY);

policy.setScanInterval(3);

// Make invocation on proxy
```

Setting the receive timeout

The receive timeout determines the number of milliseconds that an Artix client waits for a response before throwing a timeout exception. In addition to being settable in the configuration context, the receive timeout can also be set by Artix clients using the request context.

It is set using `setRecieveTimeout()`. The value is specified as an integer as shown in Example 268.

Example 269: *Setting the Timeout Interval in a Client*

```
IONAMessageContext cont = (IONAMessageContext)
  DispatchLocals.getCurrentMessageContext();

ConntectionPolicyType policy = (ConnectionPolicyType)
  cont.getRequstContext(ContextConstants.FTP_CONNECTION_POLICY);

policy.setRecieveTimeout(60000);

// Make invocation on proxy
```

Setting the Connection Credentials

Overview

FTP servers require you to connect using a username and password. These are set using the FTP connection credentials property.

Because the username and password used to connect to the FTP server must be known before the 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 398.

Setting the FTP connection credentials

To set the FTP connection credentials property you use the `ContextConstants` member `FTP_CREDENTIALS`. You are returned a `CredentialsType` object that has four methods:

- `setName()` sets a `String` representing the username used when connecting to the FTP server.
- `getName()` returns a `String` representing username used when connecting to the FTP server.
- `setPassword()` sets a `String` representing the password used when connecting to the FTP server.
- `getPassword()` returns a `String` representing the password used when connecting to the FTP server.

Example

Example 270 shows how to set the FTP connection credentials properties on an Artix FTP client.

Example 270: *Setting the FTP Connection Credentials*

```
1 ContextRegistry registry = bus.getContextRegistry();

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

Example 270: *Setting the FTP Connection Credentials*

```
3  ContextContainer cnt = registry.getConfigurationContext(name,
                                                           "SoapPort",
                                                             true);

4  CredentialsType creds =(CredentialsType)cnt.getContext(
                     ContextConstants.FTP_CREDENTIALS,
                     true);

5  creds.setUsername("george");
   creds.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 270 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 FTP credentials.
5. Set the username and password.
6. Creates the service proxy.

Setting the Coordination Policies

Overview

The FTP coordination policies determine how Artix names the files created for the messages sent over the FTP transport and how Artix cleans up files on the remote datastore. These behaviors are controlled by a set of Java classes that you can implement to meet specific needs. Artix also provides default implementations. For more information see the FTP chapter in Bindings and Transports, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Artix uses two contexts to set the naming polices. One is used for setting the naming policies for an Artix client. The other is used for setting the naming policies for an Artix server.

Because the client and server naming policies are interdependent, you need to establish the policies when the connection is initialized. Therefore it can only be set 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 398.

Setting the client-side naming policies

To set the FTP client-side naming policies you use the `ContextConstants` member `FTP_CLIENT_NAMING_POLICY`. You are returned a `ClientNamingPolicyType` object that has four methods:

- `setFilenameFactory()` sets a `String` representing the fully qualified classname of the class that implements `com.iona.jbus.transports.ftp.client.FilenameFactory`.
- `getFilenameFactory()` returns a `String` representing the name of the class of the class that implements the client-side filename factory.
- `setReplyFileLifecycle()` sets a `String` representing the fully qualified classname of the class that implements `com.iona.jbus.transports.ftp.client.ReplyFileLifecycle`.
- `getReplyFileLifecycle()` returns a `String` representing the name of the class that implements the logic for cleaning up reply files from the remote data store.

Setting the server-side naming policies

To set the FTP server-side naming policies you use the `ContextConstants` member `FTP_SERVER_NAMING_POLICY`. You are returned a `ServerNamingPolicyType` object that has four methods:

- `setFilenameFactory()` sets a `String` representing the fully qualified classname of the class that implements `com.iona.jbus.transports.ftp.server.FilenameFactory`.
- `getFilenameFactory()` returns a `String` representing the name of the class of the class that implements the server-side filename factory.
- `setRequestFileLifecycle()` sets a `String` representing the fully qualified classname of the class that implements `com.iona.jbus.transports.ftp.server.RequestFileLifecycle`.
- `getRequestFileLifecycle()` returns a `String` representing the name of the class that implements the logic for cleaning up request files from the remote data store.

Example

Example 271 shows how to set the FTP server-side naming policies.

Example 271: *Setting the FTP Server Naming Policy*

```
1  ContextRegistry registry = bus.getContextRegistry();

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

3  ContextContainer cnt = registry.getConfigurationContext(name,
                                                           "SoapPort",
                                                             true);

4  ServerNamingPolicyType namePol =
     (ServerNamingPolicyType)cnt.getContext(
                     ContextConstants.FTP_SERVER_NAMING_POLICY,
                     true);

5  namePol.setFilenameFactory("example.ServerNamingFactory");
   namePol.setRequestFileLifecycle("example.RequestLifecycle");
   ...
```

The code in Example 271 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 FTP naming policy.
5. Set the filename factory and request lifecycle.

Setting 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 Bindings and Transports, C++ Runtime.

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 `"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 `"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 Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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 272 shows the code for configuring a client for codeset conversion.

Example 272: *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 273 shows the code for configuring a server for codeset conversion.

Example 273: *Server i18n Properties*

```
// Java
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 17

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 use cases, 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 8.

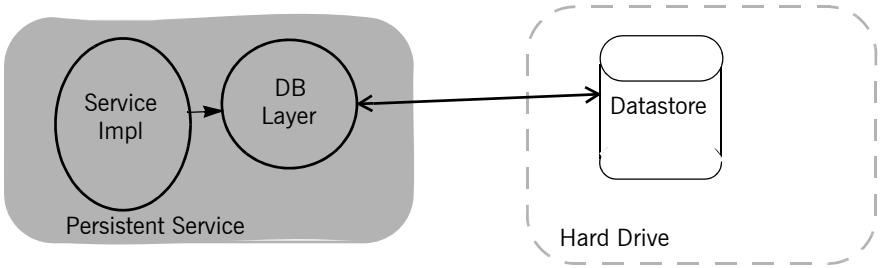

Figure 8: *The Artix Persistence Mechanism*

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 9. 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.

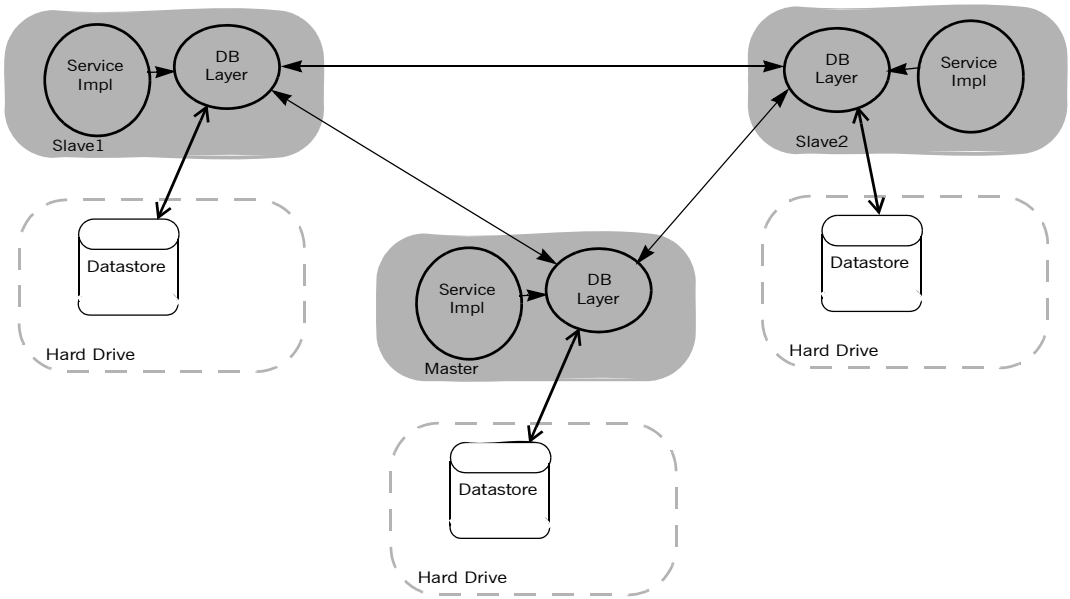

Figure 9: *Artix Service Cluster*

For more information on deploying your service as a highly available cluster see Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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 10.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.

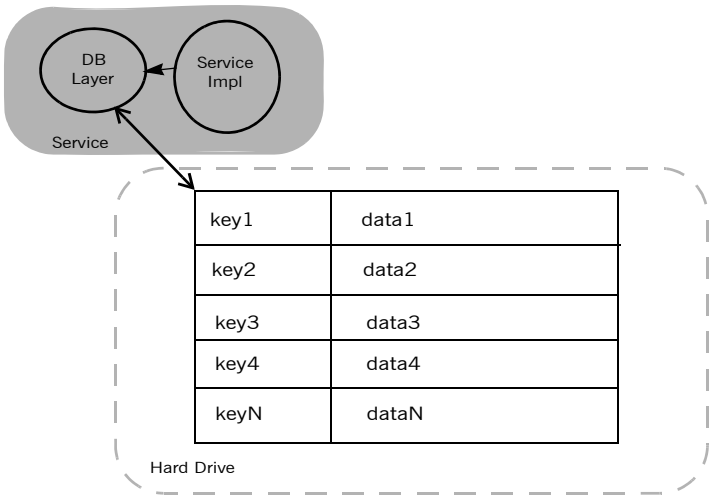

Figure 10: *Artix Persistent Datastores*

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 473. 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 then you may want to make your datastore an instance of `PersistentList`. If you want the data to be keyed using strings, 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 274.

Example 274: *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 XML Schema type that defines the class.
- The `Class` object for the class.
- The location of the contract in which the type is defined.

Example 275 shows an example of creating an `XMLDataHandler` object for the `widgetOrderInfo` type defined in Example 109 on page 184.

Example 275: *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 XML Schema 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 XML Schema definitions of the types. Objects not defined in an accessible XML Schema will cause an exception to be thrown.

Example 276 shows how to instantiate a `PersistentMap` that stores objects as XML. The id of the created datastore is `widget_table`.

Example 276: *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 277 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 277: *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 278 shows how to instantiate a `StringSerialPersistentMap` that stores `Float` objects as data. The id of the created datastore is `float_table`.

Example 278: *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 XML Schema 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 XML Schema definitions of the type the object represents. Objects not defined in an accessible XML Schema 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 279 shows how to instantiate a `StringXMLPersistentMap`. The id of the created datastore is `widget_table`.

Example 279: *Instantiating a StringXMLPersistentMap*

```
import com.iona.jbus.db.collections.*;

String wsdlPath = "file:/../widgets.wsdl";

QName dataName = new QName("http://widgets.com/widgetTypes", "widgetOrderInfo");
```

Example 279: *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 XML Schema 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 XML Schema definitions of the type. Objects not defined in an accessible XML Schema will cause an exception to be thrown.

Example 280 shows how to instantiate a `PersistentList` that stores objects as XML. The id of the created datastore is `widget_list`.

Example 280: *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 281 shows how to instantiate a `SerialPersistentList` that stores `Float` objects as data. The id of the created datastore is `float_list`.

Example 281: *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 and 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 few things to consider when using them:

- `Iterator` objects are implemented using Berkeley DB cursors that aquire a read lock on the datastore. This lock is not released 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 282 shows the code for adding an entry to a `StringSerialPersistentMap` using `put()`.

Example 282: *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 283 shows code for removing an object from a map.

Example 283: *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 key 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 284 shows code for getting an object from a map.

Example 284: *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 285 shows code for iterating through a map.

Example 285: *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 286 shows code for closing a persistent map.

Example 286: *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 few things to consider when using them:

- `Iterator` objects are implemented using Berkeley DB cursors that aquire a read lock on the datastore. This lock is not released 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 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 287 shows an example of adding an element to the end of a persistent list.

Example 287: *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 288 shows an example of removing an element from a persistent list.

Example 288: *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 288 shows an example of getting an element from a persistent list.

Example 289: *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 290 shows code for iterating through a list.

Example 290: *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 286 shows code for closing a persistent list.

Example 291: *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.

Supporting High-Availability

Overview

If you are going to use persistent maps in conjunction with the high availability features of Artix, it is necessary to perform some additional programming tasks to support *write-request forwarding*. Essentially, you must write a few lines of code to tell Artix which WSDL operations need to write to the database.

Write-request forwarding

The high availability model in Artix mirrors the high availability features of Berkeley DB. In this model, a replicated cluster consists of a master replica and any number of slave replicas. The master replica can perform both read and write operations to the database, but the slaves can perform only read operations.

What happens, though, if a client sends a write-request to one of the slave replicas? In this case, the slave replica needs to have some way of forwarding the write-request to the master replica. Artix supports this write-request forwarding feature using the `request_forwarder` plug-in on the server side. To enable the write-request forwarding feature, you must appropriately configure the server replicas, as described in Configuring and Deploying Artix Solutions, C++ Runtime, and you must perform some programming steps, as described here.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Enabling write-request forwarding

To enable your service to perform write-request forwarding you must do the following before handing control over to the bus:

1. Create a `com.iona.jbus.db.DatabaseConfig` object from the `DatabaseManager` object.
2. Create an array of strings that contains the names of all of the operations defined in the `portType` element of the service's contract that make changes to a persistent datastore.
3. Call the `DatabaseConfig` object's `markAsWriteOperations()` method to enable write-request forwarding for the specified operations.

There are three recommended places you can add this code to your service:

- In the server mainline before you call `Bus.run()`.
- In the service plug-in's `busInit()` method.
- In the creator for your service implementation object.

The demo provided with Artix uses the last method.

Creating a DatabaseConfig object

The `DatabaseConfig` object stores the configuration information used to connect to the Berkeley DB instance behind the datastore. It is created from the `DatabaseManager` object using the `getConfiguration()` method as shown in Example 292.

Example 292: *Creating a DatabaseConfig Object*

```
// Bus object bus obtained earlier
DatabaseManager mgr = new DatabaseManager(bus);
DatabaseConfig cfg = mgr.getConfiguration();
```

Creating the operation list

The write-request forwarding mechanism in Artix uses the service's contract to determine which operations to forward. In order to do this, the method used to set up the write-request forwarding takes an array of strings that contain the names of the operations for which write-requests to forward to the master.

The strings used to populate the array match the values of the `name` attribute of the `operation` elements whose implementation requires the modification of a persistent datastore. For example, imagine a service with the interface defined in Example 293.

Example 293: *Interface that Modifies Persistent Data*

```
<portType name="empService">
  <operation name="is_registered_employee">
    <input message="tns:is_registered_employee_request"
           name="is_registered_employee_request"/>
    <output message="tns:is_registered_employee_response"
            name="is_registered_employee_response"/>
  </operation>
```

Example 293: *Interface that Modifies Persistent Data*

```
  <operation name="add_employee">
    <input message="tns:add_employee_request"
           name="add_employee_request"/>
    <output message="tns:add_employee_response"
            name="add_employee_response"/>
  </operation>
  <operation name="delete_employee">
    <input message="tns:delete_employee_request"
           name="delete_employee_request"/>
    <output message="tns:delete_employee_response"
            name="delete_employee_response"/>
  </operation>
</portType>
```

The service, `empService`, defines three operations: `is_registered_employee`, `add_empolyee`, and `delete_employee`. However, only two of the operations, `add_employee` and `delete_employee`, require modifying the persistent data store. Therefore, you would place `add_employee` and `delete_employee` in the array of strings that configures the write-request forwarding mechanism as shown in Example 294.

Example 294: *Populating the Operation List*

```
String[] writeOps = { "add_employee", "delete_employee" };
```

This string will configure the write-forward request mechanism to only forward requests when a client invokes `add_employee` or `delete_employee`. If a client invokes either `add_employee` or `delete_employee` on a slave instance of this service the request will be automatically forwarded to the master instance. If a client invokes `is_registered_employee`, the slave instance will handle the request.

Marking the write operations

The last step in setting-up write-request forwarding is to mark the operations that write data to the persistent datastore. This informs Artix which operations will be forwarded to the master service instance.

You do this using the `DatabaseConfig` object's `markAsWriteOperations()` method. Its signature is shown in Example 295.

Example 295: *markAsWriteIOperations() Signature*

```
void markAsWriteOperations(String[] operations, QName service,
                           String portName, String wsdlURL);
```

It has the following parameters:

`operations`	The array of strings containing the name of the operations to mark.
`service`	The QName of the `service` element defining the endpoint.
`portName`	The value of the `name` attribute of the `port` element defining the endpoint's contact details
`wsdlURL`	The URL of the contract defining the service.

Example 296 shows the code for marking a set of operations as write operations.

Example 296: *Marking Operations for Write-Request Forwarding*

```
QName service = new QName("http://www.iona.com/persist_demo",
                          "empSOAPService");
// Bus object bus obtained earlier
DatabaseManager mgr = new DatabaseManager(bus);
1 DatabaseConfig cfg = mgr.getConfiguration();

2 String[] writeOps = { "add_employee", "delete_employee" };

3 cfg.markAsWriteOperations(writeOps, service, "empSOAPPort",
                          ".\epmServ.wsdl");
```

Configuring write-request forwarding

In order for a service to use the Artix write-request forwarding mechanism it must be configured to load the `request_forwarder` plug-in. In addition, the `request_forwarder` plug-in must be placed on the service's request interceptor list. To do this do the following:

1. Add `request_forwarder` to the list of plug-ins in the service's `orb_plugins` variable.
2. Add `request_forwarder` to the list of plug-ins in the service's `binding:artix:server_request_interceptor_list` variable.

Example 297 shows a sample configuration for a service that has write-request forwarding enabled.

Example 297: *Configuration for Write-Request Forwarding*

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

  binding:artix:server_request_interceptor_list=
   "request_forwarder";
}
```

For more information on Artix configuration see Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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 298 shows a configuration fragment for a service using persistent datastores.

Example 298: *Persistent Datastore Configuration*

```
# Artix Configuration File
...
foo_service {
    plugins:artix:db:env_name = "myDB.env";
    plugins:artix:db:home = "/etc/dbs/foo_service";
};
```

More information

For more information on Artix configuration see Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

CHAPTER 18

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 31 are not supported by the Artix implementation of the `Service` interface.

Table 31: *Unsupported Service Methods*

Method Signature
`TypeMappingRegistry getTypeMappingRegistry();`
`HandlerRegistry getHandlerRegistry();`
`Remote getPort(Class intfc) throws ServiceException;`

Table 31: *Unsupported Service Methods*

Method Signature
`Iterator getPorts() throws ServiceException;`

- The methods listed in Table 32 are not supported by the Artix implementation of the `ServiceFactory` interface.

Table 32: *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 261.

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 264.
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 299 shows the code for creating a print request and invoking on the returned print service.

Example 299: *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 299: *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 299 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 19

Instrumenting a Service

Artix provides two mechanisms that allow you to instrument your service implementations to be managed using any JMX console.

In this chapter

This chapter discusses the following topics:

Overview of Artix Instrumentation

Default instrumentation

Artix exposes a number of its internal components through a JMX complaint MBean server as shown in Figure 11. The instrumented components can be managed using any JMX console.

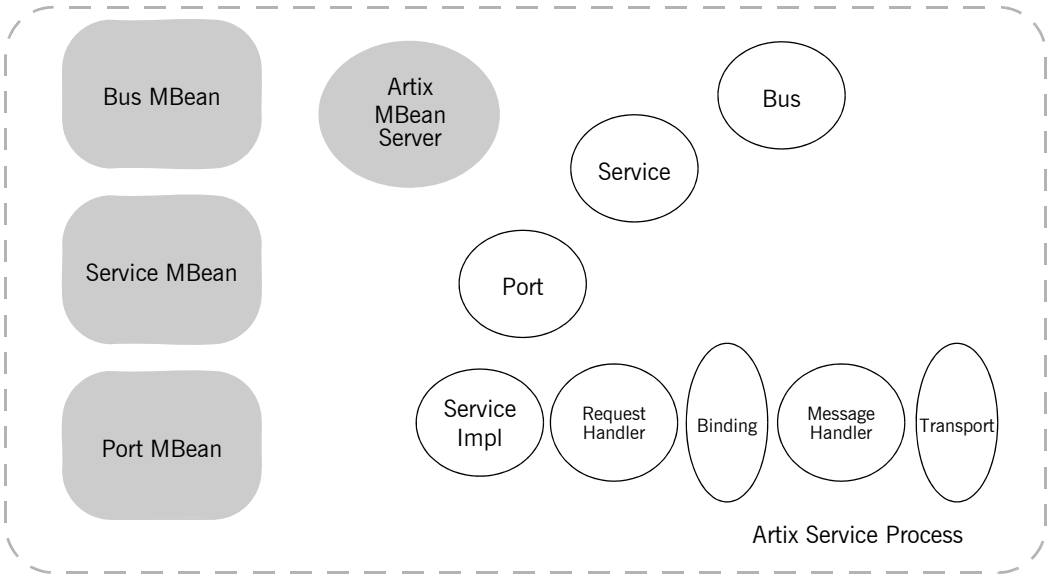

Figure 11: *Default Artix MBean Structure*

Artix Bus MBean

Each instance of an Artix bus has an MBean associated with it. The bus' MBean exposes the following properties:

- the bus identifier.
- the bus' configuration scope.
- the list of arguments passed to the bus.
- the list of service objects being managed by the bus.

The bus MBean exposes the following operations:

- enable/disable performance logging for a service.
- set/retrieve the logging level for the different Artix subsystems.

Artix Service MBean

When Artix loads a service contract, it creates a `Service` object for each `service` element in the contract. Each `Service` object has an MBean associated with it. The name of each service MBean is received from the bus' MBean.

The service's MBean exposes the following properties:

- the QName, specified in the service contract, of the service element represented by the `Service` object.
- the status of the service.
- a list of all the ports exposing this service.
- a number of service counters including:
  - the average response time of the service.
  - the total number of requests processed.
  - the total number of oneway requests processed.
  - the number errors encountered by the service.

Artix Port MBean

When Artix activates a service it creates a `Port` object for each `port` element in the activated service's `service` element. Each `Port` object has an MBean associated with it. The name of a port's MBean is received from the MBean of the `Service` object that manages the port.

The port's MBean exposes the following properties:

- the port's name as specified in the service's contract.
- the endpoint address.
- the transport's name.
- the list of message handlers through which messages on this port pass.

Additional Components

A number of other Artix provided components also provide JMX instrumentation including:

- the Artix locator.
- the Artix session manager.

- the HTTP transport.

For more information on accessing the properties exposed by Artix see Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Adding custom instrumentation

Artix allows the registration of additional MBeans with the Artix MBean server. This makes it possible for you to add custom instrumentation to your service implementations and mange it through the same management console as the other Artix components.

There are two methods of instrumenting your service implementations:

- implement one of the JMX MBean interfaces and register it with Artix's MBean server.
- implement an Artix `ManagedComponent` interface.

Functionally there is no different between the two approaches. The decision on which to use depends on ease of development, maintainability, and portability.

Activating Artix management

In order to manage the instrumented Artix components you need to add the following to your service's configuration scope:

```
plugins:bus:management:enabled="true";
```

For more information about Artix configuration see the Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Using the JMX APIs

Overview

The Artix MBean server can be accessed through the Artix bus and allows for the registration of user developed MBeans. This allows you to instrument your service implementation by developing a custom MBean using one of the JMX MBean interfaces and registering it with the Artix MBean server. Your custom instrumentation will then be accessible through the same JMX connection as the Artix internal components used by your service.

Creating your custom MBean

When you use the JMX APIs to instrument your service implementation, you follow the design methodology laid out by the JMX specification. This involves the following steps:

1. Decide what type of MBean you wish to use.
   - standard MBeans expose a management interface that is defined at development time.
   - dynamic MBeans expose their management interface at run time.
2. Create the MBean interface to expose the properties and operations used to manage your service implementation.
   - standard MBeans use the `MBean` interface.
   - dynamic MBeans use the `DynamicMBean` interface.
3. Implement the MBean class.

For example, if you wanted to add instrumentation to the widget ordering service, defined in Example 157 on page 252, that tracked the number of orders placed and average time it takes for an order to be processed. You could do this by creating a standard MBean that exposed the following attributes:

- NumOrders
- AvgTime

Note: The default instrumentation provided with Artix can provide you with statistics for the service as a whole, but to get statistics on the operations you need to add custom instrumentation.

Example 300 shows the interface for the MBean.

Example 300: *Widget Monitoring MBean Interface*

```
public interface widgetMonitorMBean
{
  public int getNumOrders();
  public int getAvgTime();
}
```

Example 301shows the class that implements the MBean.

Example 301: *Widget Monitoring MBean*

```
public class widgetMonitor implements widgetMonitorMBean
{
  int numOrders = 0;
  int avgTime = 0;

  public int getNumOrders()
  {
    return numOrders;
  }
  public void setNumOrders(int i)
  {
    numOrders = i;
  }

  public int getAvgTime()
  {
    return avgTime;
  }
  public void setAvgTime(int i)
  {
    avgTime = i;
  }
}
```

The attributes, `NumOrders` and `AvgTime`, exposed by the MBean are only readable from a management console because the interface only defines their getter methods. The service implementation can use the setter methods to update the values of the attributes.

Registering the MBean

For your MBean to be exposed to a management console, it must be registered with the Artix MBean server. The Artix MBean server is accessible through the bus' registry. Typically this will be done when your service is initialized. For services that are deployed in an Artix container, you would register you MBean in the service's `busInit()` method. For a standalone service, you would register your MBean in the service's `main()` method before calling `bus.run()`.

To register a custom MBean do the following:

1. Instantiate your custom MBean.
2. Get an instance of the bus' registry using bus.getRegistry().
3. Get the Artix MBean server from the registry using the registry's `getEntry()` method as shown in Example 302.

Example 302: *Getting the Artix MBean Server*

```
Object obj = registry.getEntry(ManagementConstants.MBEAN_SERVER_INTERFACE_NAME);
```

4. Cast the returned `Object` object into an `MBeanServer` object.
5. Create an `ObjectName` object for your MBean.
6. Register the MBean with the MBean server using the server's `registerMBean()` method.

Example 303 shows code for registering a custom MBean with the Artix MBean server.

Example 303: *Registering a Custom MBean*

```
import javax.management.*;
import com.iona.jbus.management.ManagementConstants;

1  widgetMonitor widgetMon = new widgetMonitor();

   Bus bus = getBus();

2  BusRegistry registry = bus.getRegistry();

3  Object obj =
      registry.getEntry(ManagementConstants.MBEAN_SERVER_INTERFACE_
      NAME);
```

Example 303: *Registering a Custom MBean*

```
4  MBeanServer mbeanServer = (MBeanServer)obj;

5  ObjectName name = new OBjectName("WidgetOrderMonitor");

6  mbeanServer.registerMBean(widgetMon, name);
```

Using the Artix ManagedComponent Interface

Overview

If you do not want to use the JMX interfaces to add custom instrumentation to your service, you can use the Artix `ManagedComponent` interface. This interface wraps the JMX subsystem in proprietary interfaces. You do not need to access the Artix JMX server to add instrumentation to your service.

Procedure

To add custom instrumentation to your service using the `ManagedComponent` interface you need to do the following:

1. Implement an instrumentation class that implements both the `com.iona.jbus.managment.ManagedComponent` interface and the `com.iona.jbus.management.Instrumentation` interface.
2. Implement a support class for your instrumentation that implements the `com.iona.jbus.management.MBeanInfoGenerator` interface.
3. In the service's initialization routine, instantiate your instrumentation object and register it with the bus.
4. In the service's shutdown routine, unregister your instrumentation object.

Note: If your service is designed to be a standalone service, you do not need to do step **4**.

In this section

This section discusses the following topics:

Implementing the Instrumentation Class

Overview

Like an MBean, a `ManagedComponent` style instrumentation class is responsible for providing access to the attributes you want to track and any management operations you want to expose.

Unlike an MBean, you do not need to define an interface for your instrumentation class. Instead, your instrumentation object implements two Artix management interfaces and defines the operations required to expose the attributes and operations you want.

Implementing the creator

Your instrumentation class must have at least one public contrusctor that takes no arguments. You can use the default constructor provided by Java to fulfil this requirement.

Interfaces to implement

Your instrumentation class needs to implement the following Artix management interfaces:

- `com.iona.jbus.management.ManagedComponent`
- `com.iona.jbus.management.Instrumentation`

The `Instrumentation` interface is a marker interface that has no methods that need to be implemented.

The `ManagedComponent` interface is the wrapper that allows the Artix runtime to extract management information from your service. It has three methods that need to be implemented: `getInstrumentation()`, `getObjectName()`, and `setObjectName()`.

Implementing the ManagedComponent methods

You must provide the implementation for the three operations defined in the `ManagedComponent` interface. For most applications, the implementations for these operations can be boilerplate.

getInstrumentation()

`getInstrumentation()` is called by the bus to obtain an instance of your instrumentation class. Its signature is shown in Example 304.

Example 304: *getInstrumentation()*

```
public Instrumentation getInstrumentation();
```

For most cases, this method can simply return an instance of itself as shown in Example 305. You can, however, do other initialization work in this method.

Example 305: *Implementing getInstrumentation()*

```
public Intrumentation getInstrumentation()
{
  return this;
}
```

setObjectName()

`setObjectName()` provides a mechanism for setting the name of an instrumentation object. Its signature is shown in Example 306.

Example 306: *setObjectName()*

```
public void setObjectName(ObjectName name);
```

It takes a `javax.management.ObjectName`. If you don't wish to expose this functionality, you can implement this method to just return void. The name of the instrumentation object can be completely handled by `getObjectName()`.

getObjectName()

`getObjectName()` returns the name of an instrumentation object. Its signature is shown in Example 307.

Example 307: *getObjectName()*

```
public ObjectName getObjectName();
```

It returns a `javax.management.ObjectName`. The name returned is the name by which an instrumentation object is identified by a JMX console.

Defining attributes and operations

Your instrumentation class is responsible for providing the methods used to get and set the attributes exposed by your instrumentation. It is also responsible for providing the methods used to implement any operations exposed by your instrumentation. The methods to do this is not part of any of the Artix management interfaces.

The naming pattern for attributes and operations follow the same patterns as those used by MBeans. Each attribute must have at least a setter or a getter method. The setter methods use the naming pattern `set`*`AttributeName`*`()`. The getter methods use the naming pattern `get`*`AttributeName`*`()`. Operations can have any name you would like.

So, if you wanted to expose the attributes `NumOrders` and `AvgTime` you would implement the following methods:

- `getNumOrders()`
- `setNumOrders()`
- `getAvgTime()`
- `setAvgTime()`

Example

Example 308 shows a fully implemented instrumentation class.

Example 308: *ManagedComponent Style Instrumentation Object*

```
import com.iona.jbus.management.Intrumentation;
import com.iona.jbus.management.ManagedComponent;
import com.iona.jbus.management.ManagementException;

import javax.management.ObjectName;

public class WidgetManager
implements ManagedComponent, Instrumentation
{
  private ObjectName objName;
  private int numOrders;
  private float avgTime;
  private float totalTime;

  public WidgetManager()
  {
     objName = new ObjectName("WidgetManager");
     numOrders = 0;
     avgTime = 0.0;
     totalTime = 0.0;
  }

  public Instrumentation getInstrumentation()
  {
    return this;
  }
```

Example 308: *ManagedComponent Style Instrumentation Object*

```
  public void setObjectName(ObjectName name)
  {
    return;
  }

  public ObjectName getObjectName()
  {
    return objName;
  }

  public void setNumOrders(int num)
  {
  numOrders += num;
  }

  public int getNumOrders()
  {
    return numOrders;
  }

  public void setAvgTime(float time)
  {
    totalTime += time;
    avgTime = totalTime/numOrders;
  }

  public float getAvgTime()
  {
    return avgTime;
  }
}
```

Implementing the Support Class

Overview

Under the covers, a `ManagedComponent` style instrumentation class is used by Artix to generate a `ModelMBean` that is used by the Artix management infrastructure. To facilitate the `ModelMBean` generation, you are required to provide a support class for your instrumentation class. This support class is responsible for creating a `ModleMBeanInfo` object that describes your instrumentation class.

Naming convention

Your support class must use the following naming convention to be recognized by Artix:

```
instrumentationClassNameSupport
```

For example, if your instrumentation class is named `WidgetManager`, the corresponding support class must be named `WidgetManagerSupport`.

Interface to implement

Your support class needs to implement the Artix interface `com.iona.jbus.management.MBeanInfoGenerator`. It has one public method, `getModel()`, that you need to implement.

`getModel()` is called by Artix when it creates that MBean for your instrumentation. As shown in Example 309, it returns a `javax.management.ModelMBeanInfo` object that fully describes the attributes and operations exposed by your instrumentation class.

Example 309: *getModel()*

```
public ModelMBeanInfo getModel(ObjectName objName,
                               String displayName)
throws RuntimeOperationsException, MBeanException;
```

For more information on populating a `ModelMBeanInfo` object see http://java.sun.com/products/JavaManagement.

Example

Example 310 shows the support class for the instrumentation class defined in Example 308 on page 518.

```
import com.iona.jbus.management.MBeanInfoGenerator;
import java.lang.reflect.Constructor;
import javax.management.Descriptor;
import javax.management.MBeanException;
import javax.management.ObjectName;
import javax.management.RuntimeOperationsException;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanConstructorInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanNotificationInfo;
import javax.management.modelmbean.ModelMBeanOperationInfo;

public class WidgetManagerSupport implements MBeanInfoGenerator
{
  public WidgetManagerSupport()
  {
  }

  protected ModelMBeanAttributeInfo[] getAttributes()
  {
    ModelMBeanAttributeInfo[] attributes =new ModelMBeanAttributeInfo[2];

    Descriptor orderDescriptor = new DescriptorSupport(new String[]
      {
        "name=NumOrders",
        "class=WidgetManager",
        "descriptorType=attribute",
        "getMethod=getNumOrders",
        "setMethod=setNumOrders",
        "value=0",
        "default=0",
        "displayName=Number of orders processed",
        "persistPolicy=NoMoreOftenThan",
        "persistPeriod=300",
        "currencyTimeLimit=0",
        "persistLocation=/data",
        "persistName=WidgetManager.ser",
        "default=0"
      });
```

```
    Descriptor timeDescriptor = new DescriptorSupport(new String[]
      {
       "name=AvgTime",
       "class=WidgetManager",
       "descriptorType=attribute",
       "getMethod=getAvgTime",
       "setMethod=setAvgTime",
       "value=0",
       "default=0",
       "displayName=Average time to process an order",
       "persistPolicy=NoMoreOftenThan",
       "persistPeriod=300",
       "currencyTimeLimit=0",
       "persistLocation=/data",
       "persistName=WidgetManager.ser",
       "default=0"
      });

    attributes[0] = new ModelMBeanAttributeInfo("NumOrders",
                                                "java.lang.Integer",
                                                "",
                                                true,
                                                true,
                                                false,
                                                orderDescriptor);

    attributes[1] = new ModelMBeanAttributeInfo("AvgTime",
                                                "java.lang.Float",
                                                "",
                                                true,
                                                true,
                                                false,
                                                timeDescriptor);

    return attributes;
  }

  protected ModelMBeanOperationInfo[] getOperations()
  {
    ModelMBeanOperationInfo[] operations = new ModelMBeanOperationInfo[0];
    return operations;
  }
```

```
  protected ModelMBeanNotificationInfo[] getNotifications()
  {
    ModelMBeanNotificationInfo[] notifications = new ModelMBeanNotificationInfo[0];
    return notifications;
  }

  public ModelMBeanInfo getModel(ObjectName objName, String displayName)
  throws RuntimeOperationsException, MBeanException
  {
    Descriptor modelDescriptor = new DescriptorSupport(new String[]
      {
        "name=WidgetManagerBean",
        "descriptorType=mbean"
      });

    ModelMBeanConstructorInfo[] constructors = new ModelMBeanConstructorInfo[0];

    ModelMBeanInfo model = new ModelMBeanInfoSupport("WidgetManager",
                                                     "Widget Sales mbean",
                                                     getAttributes(),
                                                     constructors,
                                                     getOperations(),
                                                     getNotifications());

    model.setMBeanDescriptor(modelDescriptor);

    return model;
  }
}
```

Creating and Removing your Instrumentation

Overview

To make your custom instrumentation available to management consoles, you must create an instance of your instrumentation class. Then you need to tell the bus to create an MBean for your instrumentation. The bus automatically registers the MBean with the Artix JMX server.

Unlike JMX-style instrumentation, `ManagedComponent`-style instrumentation must be cleaned up. In your services shutdown() routine you need to tell the bus to remove the MBean created for your instrumentation. This also cleans up any other resources created to support the custom instrumentation.

Creating the instrumentation

As with JMX-style instrumentation `ManagedComponent`-style instrumentation is not available until an MBean is created and registered with the Artix MBean server. However, when you create `ManagedComponent`-style instrumentation you do not directly create an MBean or register it with the MBean server. This is all handled by the bus.

To create an MBean for your instrumentation and register it with the MBean server do the following:

1. Instantiate an instance of your instrumentation class.
2. Instantiate a `ManagedComponentEvent` using `ManagedComponentCreateEvent()`.
3. Send the event to the bus using `Bus.sendEvent()`.

Example 310 shows code for creating an MBean for your custom instrumentation.

Example 310: *Creating a MBean for a Managed Component*

```
public class WidgetPlugin extends BusPlugIn
{
  WidgetManager inst;

  public void busInit() throws BusException
  {
    Bus bus = getBus();

1     inst = new WidgetManager();
2     ManagedComponentEvent create = new
     ManagedComponentCreatedEvent(inst);
```

Example 310: *Creating a MBean for a Managed Component*

```
    bus.sendEvent(create);
    ...
  }
...
}
```

3

Removing your instrumentation

To clean up your custom instrumentation you need to unregister the MBean created to support it and destroy the MBean. This is all done using a bus event.

To remove your custom instrumentation from the JMX serer do the following:

1. Instantiate a `ManagedComponentEvent` using `ManagedComponentRemovedEvent()`.
2. Send the event to the bus using `Bus.sendEvent()`.

Example 311 shows code for creating an MBean for your custom instrumentation.

Example 311: *Removing the MBean for a Managed Component*

```
public class WidgetPlugin extends BusPlugIn
{
  WidgetManager inst;

  ...

  public void busShutdown() throws BusException
  {
    Bus bus = getBus();

    ManagedComponentEvent create = new
   ManagedComponentRemovedEvent(inst);
    bus.sendEvent(create);
    ...
  }
}
```

1

2

CHAPTER 20

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:

Understanding the Artix Plug-in Model

In this section

This section discusses the following topics:

Artix plug-ins

An *Artix plug-in* is a well-defined component that can be independently loaded into an Artix application. Artix defines a platform-independent framework for loading plug-ins dynamically, based on the dynamic linking capabilities of modern operating systems.

Plug-ins, due to the platform-independent nature of Artix, can be implemented in either C++ or Java and be loaded into any Artix application. Plug-ins developed in Java are packaged as independent JAR files that are located by Artix using configuration information. Java based plug-ins can be loaded into Artix applications developed in C++.

Configuration

The plug-ins that an application should load are specified by the `orb_plugins` configuration variable, which contains a list of plug-in names.

In addition, for each plug-in that is to be loaded, 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`.

For example, the following extract shows how to configure an application, whose ORB name is `plugin_example`, to load a single plug-in, `sample_artix_interceptor`.

```
# Artix domain configuration file
...
plugin_example {
    orb_plugins = ["sample_artix_interceptor"];

    plugins:sample_artix_interceptor:classname =
   "samplePlugInFactory";
};
```

Loading the plug-in

Figure 12 shows how a plug-in is loaded as the application starts up. The steps to load the plug-in are as follows:

1. The user launches the application, `app`, specifying the bus name as `plugin_example` at the command line.
2. As the application starts up, it scans the Artix configuration file to determine which plug-ins to load. Priority is given to the configuration settings in the `plugin_example` configuration scope.
3. The Artix core loads the plug-ins specified by the application's configuration.

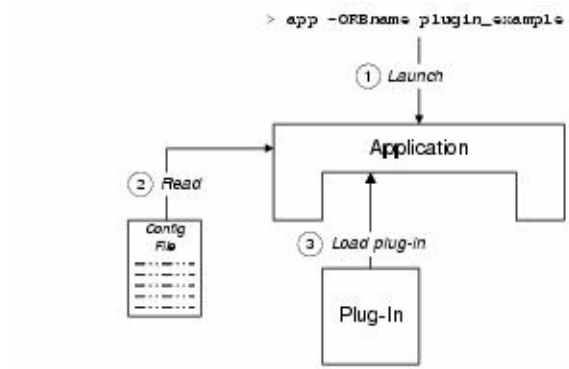

Figure 12: *Loading a Plug-In*

Initializing the plug-in

Plug-ins are usually initialized when the bus is initialized. Figure 13 shows the plug-in initialization sequence, which proceeds as follows:

1. The bus is initialized.
2. The Artix core iterates over all of the plug-ins in the `orb_plugins` list, calling `BusPlugInFactory.createBusPlugin()` on each one.
3. The `BusPlugInFactory` object creates a `BusPlugIn` object, which initializes the state of the plug-in for the current bus instance.
4. After all of the `BusPlugIn` objects have been created, the Artix core calls `busInit()` on each `BusPlugIn` object.

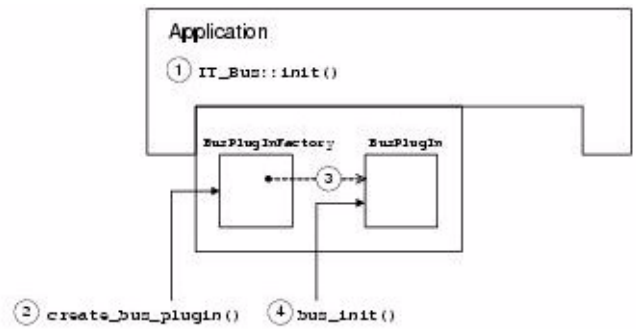

Figure 13: *Initializing a Plug-In*

BusPlugInFactory object

A `BusPlugInFactory` object provides the basic hook for initializing an Artix plug-in. A single static instance of the `BusPlugInFactory` object is created when the plug-in is loaded into an application. See "Implementing the BusPlugInFactory Interface" on page 534 for more details.

BusPlugIn object

A `BusPlugIn` object caches the state of the plug-in for the current bus instance. The `BusPlugIn` object is responsible for performing most of the plug-in initialization and shutdown tasks. See "Extending the BusPlugIn Class" on page 531 for more details.

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 312 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 312: *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 313 shows a `busInit()` the registers two message handlers.

Example 313: *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();

    bus.registerHandlerFactory(new firstHandFactory());
    bus.registerHandlerFactory(new secondHandFactory2());
  }

  ...
}
```

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 service 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 314.

Example 314: *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 315 shows the signature for `createBusPlugIn()`.

Example 315: *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 316 shows the signature for `destroyBusPlugIn()`.

Example 316: *destroyBusPlugIn()*

```
public void destroyBusPlugIn(BusPlugIn plugin);
```

Example

For example, the `BusPlugInFactory` implementation for a plug-in `BankPlugIn` would look similar to Example 317.

Example 317: *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 318 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 318: *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 319.

Example 319: *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 318, you would use the configuration fragment shown in Example 320.

Example 320: *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 321 shows a configuration fragment for loading the junk message plug-in using a classloader environment.

Example 321: *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 633.

CHAPTER 21

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 14.

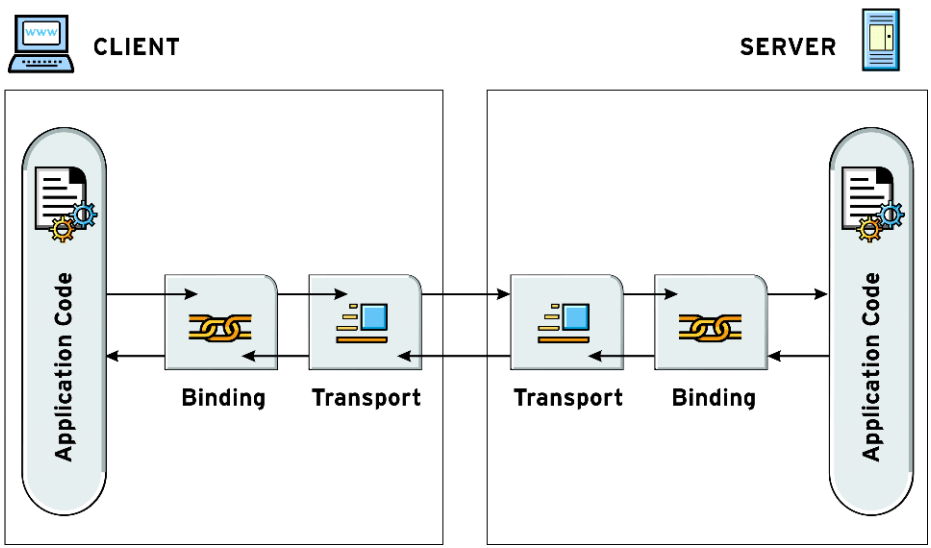

Figure 14: *The Life of a Message*

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 15.

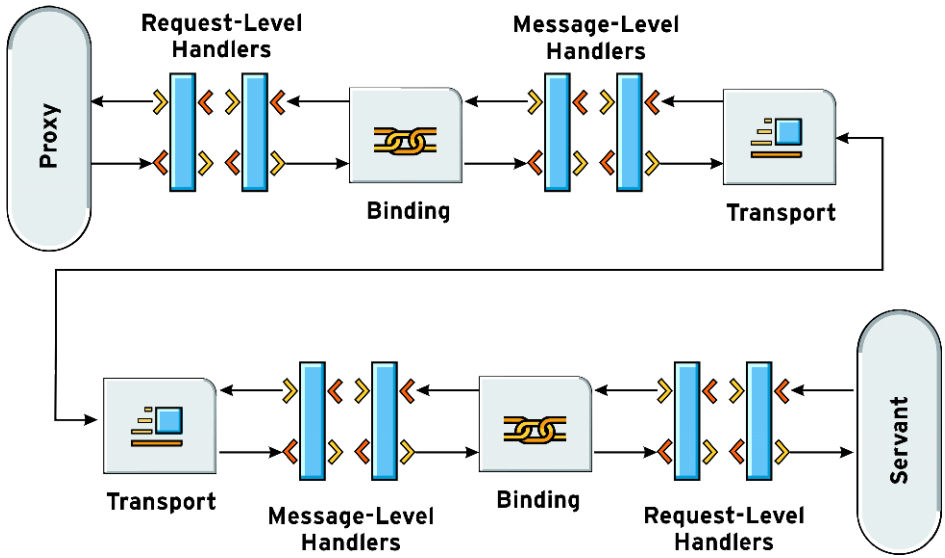

Figure 15: *Handler Levels*

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 can be implemented as standalone Java classes or as Artix plug-ins. Implementing your handlers as a standalone Java class is the simplest method. This method only requires you to implement the JAX-RPC `Handler` interface. Artix will load all of the handlers based on the endpoint's configuration scope and register them with the bus.

Implementing your handlers as plug-ins requires that you implement more classes, but it also provides you some added control over your implementation. In addition to the JAX-RPC `Handler` interface, this method requires that you implement the `BusPlugInFactory` interface, extend the `BusPlugIn` class, implement the `HandlerFactory` interface, and register the handler factory with the bus. For information about implementing handlers in a plug-in see "Implementing a Handler as a Plug-in" on page 549.

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.

Generic implementations

To simplify implementing a handler, 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.

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.

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 322. Both methods have a `MessageContext` as an argument. For information on using the message contexts see "Using Message Contexts" on page 337. The return value should reflect the state of the message processing. If the message is successfully processed return `true`. If not, return `false`.

Example 322: *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 323 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. The object used to hold the SOAP header is of the generated type `SOAPHeaderInfo`. This type is generated from a user supplied XML Schema document that describes the contents of the SOAP header. For more information see "Sending Message Headers" on page 365.

Example 323: *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.

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 324. Both methods have a `MessageContext` as an argument. For information on using the message contexts see "Using Message Contexts"

on page 337. The return value should reflect the state of the message processing. If the message is successfully processed return `true`. If not, return `false`.

Example 324: *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 578. 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 575. 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 325 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 325: *Client Message-Level Handler*

```
// Java
import com.iona.jbus.*;
com.iona.jbus.jaxrpc.handlers.GenericHandler;

import java.io.*;
import javax.xml.namespace.QName;
```

Example 325: *Client Message-Level Handler*

```
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;
  }
}
```

Implementing a Handler as a Plug-in

Overview

If you choose to implement your handlers as Artix plug-in, you need to do the following:

- Implement the Artix plug-in interfaces as described in "Developing Plug-Ins" on page 527.
- Implement the `HandlerFactory` interface for the handlers loaded by the plug-in.

In this section

This section discusses the following topics:

Creating the Handler Plug-in

Overview

Artix handlers can 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 534.
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 326.

Example 326: *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 327 shows a the plug-in code for a handler.

Example 327: *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 327 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 a handle for the plug-in's bus.
5. Registers the handlers' factories with the bus using `registerHandlerFactory()`.

6. Implements `busShutdown()`.

Creating a Handler Factory

Overview

When you implement your handler in a plug-in, the bus calls the methods provided by the `HandlerFactory` you register in the handler plug-in. You implement a `HandlerFactory` for each set of handlers you are deploying in a plug-in. 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 328. They take a single `HandlerInfo` object and return an instance of the class `HandlerInfo`.

Example 328: *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 329 shows code for implementing a handler factory.

Example 329: *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 329: *Handler Factory For Request Level Handlers*

```
  public HandlerInfo getServerRequestHandler(HandlerInfo)
  {
    info.setHandlerClass(secondHandServerRequestHandler.class);
    return info;
  }
}
```

The code in Example 329 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 330.

Example 330: *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 33.

Table 33: *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 331 shows code for getting all of the properties from a `HandlerInfo` object.

Example 331: *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);
```

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 230.
2. Create an instance of the user defined fault you want to throw. See "Working with User-defined Exceptions in Artix Applications" on page 235.
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 332.

Example 332: *Service Definition with a Fault*

```
...
<message name="pied">
  <part name="flavor" type="xsd:string" />
</message>
```

Example 332: *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 332 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 333.

Example 333: *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 334.

Example 334: *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 337.

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 569.

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 335 shows code for getting an exception in `handleFault()`.

Example 335: *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.

Configuring Endpoints to Use Handlers

Overview

Configuring an endpoint to load and use handlers is a two step process. First, you must specify the class that implements and associate it with a name. Second, you must add the handler to one of the endpoint's interceptor chains.

Specifying the implementation class

How you specify the implementation class for your handler depends on how you implemented your handler.

Handlers implemented as a Java class

If you implemented your handler as a plain Java class, you specify the implementation class using a configuration variable of the form:

```
handler:handlerName:classname="handlerClassname";
```

The value you supply for *handlerName* is the name by which the handler will be referred to in the interceptor chains. The value you supply for *handlerClassname* is the fully qualified class name of your handler's implementation. For example, if you wrote a handler for scrubbing messages in a class called `com.squeaky.ScrubberHandler` you would add the configuration variable shown in Example 336 to your endpoint's configuration.

Example 336: *Handler Class Specification*

```
handler:scrubber:classname="com.squeaky.ScrubberHandler";
```

When adding the handler to the endpoint's interceptor chain you would refer to the handler using `scrubber`.

Handlers implemented as a Plug-in

If you implemented your handler as an Artix plug-in, you specify its implementation using the method described in "Configuring Artix to Load a Plug-in" on page 536.

Adding handlers to an interceptor chain

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.

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.

Example 338 shows the configuration for an application that uses both client and server handlers.

Example 337: *Configuration with Handlers*

```
java_interceptors
  client
  {
    binding:artix:client_request_interceptor_list =
   "firstHand+secondHand";
    binding:artix:client_message_interceptor_list =
   "firstHand+secondHand";
  };
```

Example 337: *Configuration with Handlers*

```
  server
  {
    binding:artix:server_request_interceptor_list=
   "secondHand+firstHand";
    binding:artix:server_message_interceptor_list =
   "secondHand+firstHand";
     };
};
```

More information

For more information on configuring Artix applications see Configuring and Deploying Artix Solutions, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Example 338 shows the configuration for an application that uses both client and server handlers.

Example 338: *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
  };
```

Example 338: *Configuration with Handlers*

```
  server
  {
    binding:artix:server_request_interceptor_list=
   "secondHand+firstHand";
    binding:artix:server_message_interceptor_list =
   "secondHand+firstHand";

    # override config settings for server here
    };
};
```

CHAPTER 22

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 339 the operation name returned from the context would be `forward`.

Example 339: *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 340 shows the code for getting the operation name from the message context.

Example 340: *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 341.

Example 341: *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 342 shows s server-side request handler that converts the `length` part of an incoming request from inches to centimeters.

Example 342: *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 342 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 341 on page 573, `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 34.

Table 34: *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 35.

Table 35: *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 343 shows an example of using the `SOAPMessageContext` to add an attachment to a SOAP message.

Example 343: *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 343 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 344.

Example 344: *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 345. 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 345: *StreamMessageContext*

```
package com.iona.jbus;
```

Example 345: *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 346 shows code for adding a string to the end of a message.

Example 346: *Using StreamMessageContext*

```
class TestInputStream extends InputStream
{
  InputStream in;
  ByteArrayInputStream bin;

  TestInputStream(InputStream i2, byte bytes[])
    {
      in = i2;
      bin = new ByteArrayInputStream(bytes);
    }
```

Example 346: *Using StreamMessageContext*

```
  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 23

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 XML Schema 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 XML Schema document.

Procedure

To develop a custom Artix transport you need to do the following:

1. Make an XML Schema 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 597.
5. Implement the `ServerTransport` interface as shown in "Developing the Server Transport" on page 605.

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 XML Schema 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 XML Schema 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 XML Schema 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 347.

Example 347: *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 622.

When writing the XML Schema for your transport's attributes you specify the transport's namespace as the target namespace. This is done using the `targetNamespace` attribute of the XML Schema document's `schema` element, as shown in Example 348.

Example 348: *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 348, 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 XML Schema definition must conform to the following rules:

1. It must import the WSDL 1.1 XML Schema 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 349 shows an example of an XML Schema document for a transport that uses a single element, `sock:address`, to define an endpoint.

Example 349: *Sample Transport XML Schema*

```
<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 349 does the following:

1. Defines the target namespace for the transport's attributes.
2. Imports the WSDL XML Schema 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 349 you would include `xmlns:sock="http://widgetVendor.com/transport/socket"` in the contract's `definition` element and a `service` element similar to Example 350.

Example 350: *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 527.

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 36.

Table 36: *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 592.
`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 36: *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 592.

Example

Example 351 shows a transport factory for a custom transport.

Example 351: *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 351: *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 37.

Table 37: *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 351, 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 351 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 592.

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 38 explains these values.

Table 38: *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 352 shows a transport plug-in that registers and unregisters a transport factory with the bus.

Example 352: *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 352: *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 527.

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 39: *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 353.

Example 353: *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 354 shows the `initialize()` method for the custom transport demo.

Example 354: *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 354: *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 354 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 355. `connect()` is called immediately after `initialize()` and is only called once per transport instance.

Example 355: *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 356. `disconnect()` is called just before the bus destroys the resources used by the transport's plug-in.

Example 356: *disconnect()*

```
void disconnect() throws BusException
```

Example 357 shows code for making and breaking a socket connection.

Example 357: *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 357: *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 358, 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 358: *getOutputStream()*

```
OutputStream getOutputStream(MessageContext context)
throws TransportException;
```

Example 359 shows the `getOutputStream()` implementation in custom transport demo.

Example 359: *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 337.

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 360.

Example 360: *Invoking Operations From the Transport*

```
InputStream invoke(OutputStream request, MessageContext context)
throws TransportException
void invokeOneway(OutputStream request, MessageContext context)
throws TransportException
```

Example 361 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 361: *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 40.

Table 40: *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 40: *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 353.

Example 362: *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 41 shows what `activate()` needs to do for all combinations of message port threading policy settings and threading resource policy settings.

Table 41: *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 41: *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 363 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 363: *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 363: *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 363: *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 354 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 616.

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 364.

Example 364: *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 616.

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 365 shows a thread for a server transport with a threading resource policy of `USES_WORKQUEUE`.

Example 365: *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 366 shows a work item for a server transport.

Example 366: *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 367 shows the code for implementing `run()` for a multi-threaded transport.

Example 367: *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 367: *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 367 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 368, takes a message context as a parameter and returns a Java `OutputStream` into which the transport callback object will write the response.

Example 368: *ServerTransport.getOutputStream()*

```
public OutputStream getOutputStream(MessageContext ctx)
throws TransportException;
```

Example 369 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 369: *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 370, `postDispatch()` takes the `OutputStream` containing the response and the request's message context.

Example 370: *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 371: *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 337.

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 372 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 372: *Disconnecting a Custom Server Transport*

```
// Java
import iona.com.jbus.*;

public class SocketServerTransport implements ServerTransport
{
...
```

Example 372: *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 373 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 373: *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 373: *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 Bindings and Transports, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

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 374.

Example 374: *Specifying a Transport Namespace*

```
namespace:xml_namespace:plugin="plugin_name";
```

xml_namespace is the target namespace in the XML Schema 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 375.

Example 375: *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 625.

CHAPTER 24

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 Progress' 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 376 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 376: *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 376 is specified as `demo.telco.server`. During process initialization, Artix searches for a configuration scope with the same name as the `-ORBname` parameter. To specify an `-ORBname`, you use the following syntax:

```
<processName> [application parameters] -ORBname configScope
```

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 377.

Example 377: *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 378 shows a variable with a string value.

Example 378: *String Value*

```
plugins:junk:junkyard="\etc\junkyard";
```

Example 379 shows a variable with a list value.

Example 379: *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 380 shows code for getting the configuration in a plug-in.

Example 380: *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 380 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 381. 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 381: *getString()*

```
String getString(String name);
```

Example 382 shows the code for reading the variable `plugins.junk.junkyard`.

Example 382: *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 381. 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 383: *getString()*

```
String[] getlist(String name);
```

Example 382 shows the code for reading the variable `plugins.junk.filters` and printing out the values.

Example 384: *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 25

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 visible to each classloader in an application's classloader chain.

In this chapter

This chapter discusses the following topics:

Class Loading: An Overview

Introduction

Part of the mechanism that allows Java's 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 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 16, 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`, class `a1` will not need to be loaded because it is supplied by classloader A.

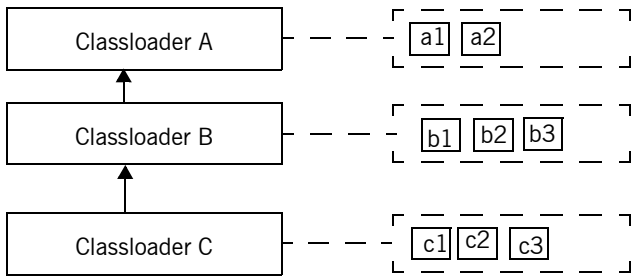

Figure 16: *Classloader Chain*

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 17.

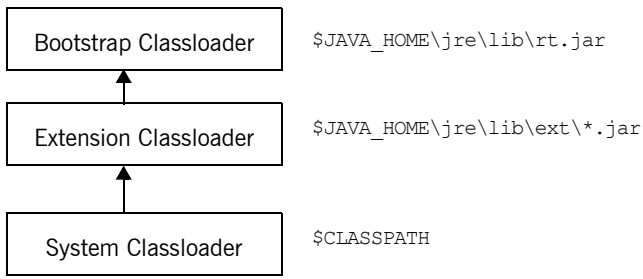

Figure 17: *Default Classloader Hierarchy*

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 that user defined classes are not isolated 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.

Classes can also become inaccessible because it is possible for a class to 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 JVM's default classloader hierachy when the bus, or any Artix plug-in, is loaded. The first additional classloader 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 specified set of resources. This is shown in Figure 18.

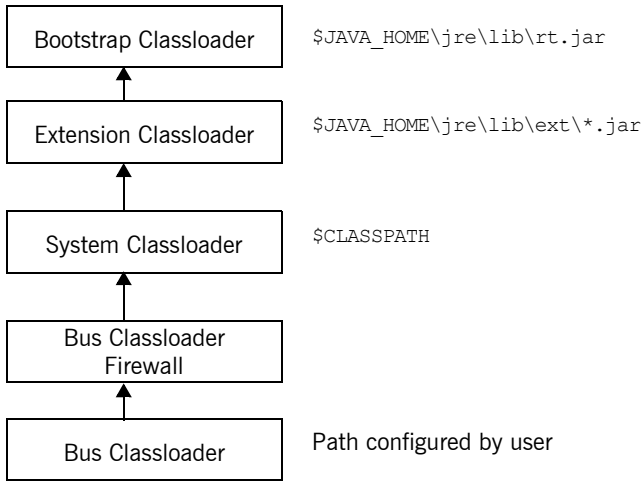

Figure 18: *Artix Bus Classloader Chain*

Why use the added classloaders?

Adding these two classloaders solves both of the problems of Java's classloader system. The classloader firewall 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. The Artix classloader will load versions of the blocked classes based on the resources specified.

It solves the problem of inaccessible classes 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 19. 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.

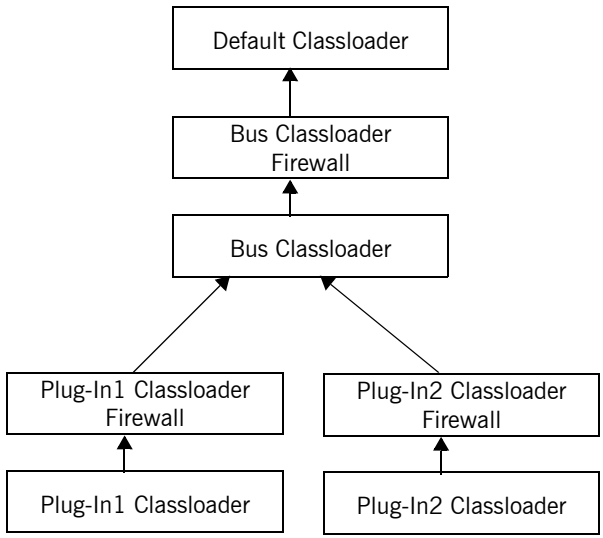

Figure 19: *Artix Plug-In Classloader Chain*

If the bus blocks a system class from the plug-ins, it creates problems for the plug-ins. Therefore you must be careful when creating the rules defining what classes are 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 a 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. Despite its power, the classloader environment 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.

Using the firewall with optimized types

Artix uses some generated code to optimize performance the performance of the runtime. This generated code is used when a user registers a `TypeFactory` with the bus. The Artix firewall classloader should not be used in conjunction with these generated classes. It is extremely difficult to create the proper filters to allow all of the generated classes through the firewall.

In order to use the firewall classloader you need to tell the Artix runtime to not use the generated classes and to fall back on dynamic runtime support. To use the firewall classloader when you have registered a `TypeHandler` with the Bus, do one of the following:

- Set a Java system property.

```
-Dgenerated_type_handler.disabled=true
```

- Set a configuration property in the Artix configuration file.

```
java:generated_type_handler:disabled=true
```

- Set a `Bus` property.

```
hashtable.put("generated_type_handler.disabled", "true");
Bus bus = Bus.init(args, hashtable);
```

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. 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 385.

Example 385: *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 file. 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 386. In addition, you can use the optional parent attribute to define the classloader's parent as discussed in "Chaining classloaders" on page 643.

Example 386: *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. The `ce:firewall` element is the first child of the `ce:environment` element and has one or more `ce:filter` child elements. Defining firewall filters is described in "Configuring the classloader firewall" on page 643.

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 the `ce:environment` element. 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 645.

Chaining classloaders

You chain a classloader 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 firewall will allow access to all classes from the location specified in the `discover-source` attribute. Table 42 shows the values for `discover-source`.

Table 42: *discover-source values for the Classloader Firewall*

Value	Meaning
`jre`	Allow access to all of the classes for the currently running JRE. It is highly recommended that this filter is included in your firewall definition.
`jar`	Allow access to all of the classes from the specified jar file. Jar file locations can be given using relative or absolute file names. For example to access all of the classes in `myApp.jar`, you could define a filter like `<ce:filter type="discover" discover-source="jar">.\myApp.jar</ce:filter>`.

Table 42: *discover-source values for the Classloader Firewall*

Value	Meaning
`jar-of`	Allow access to the 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 387 will use the value of `LIB` to resolve the specified path.

Example 387: *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 388 shows a sample CE file.

Example 388: *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">
```

Example 388: *Simple CE File*

```
    <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 389.

Example 389: *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 389 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 625 and Configuring and Deploying Artix Applications, C++ Runtime.

Note: The JAX-RPC APIs are implemented on top of the Artix ESB C++ Runtime using a JNI layer.

Index

D

I

J

L

M

O

P

R

S

T

U

W

X