
# Artix™

## Technical Use Cases

Version 4.2, March 2007

Making Software Work Together™

Updated: March 12, 2007

# Contents

# Preface

## What is Covered in this Book

This book covers a number of typical cases where Artix can be used to solve a problem. The use cases progress from the simplest, such as developing a client or a server, to more involved use cases, such as integrating transactional services involving different middlewares.

For each use case, there is a brief description of the scenario, followed by a presentation of the step by step procedure. Most steps in the process provide links to related sections of the Artix documentation library. These links provide you with detailed information about what is happening at each step.

## Who Should Read this Book

This guide is intended for all users of Artix. This guide assumes that you have a working knowledge of the middleware transports used to implement the Artix system. It also assumes that you are familiar with basic software design concepts, and that you have a basic understanding of WSDL.

## 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.

CHAPTER 1

# Building a Service Consumer

*Artix makes building a service consumer simple.*

**Overview**

The most basic use for Artix is to build a consumer of a preexisting service. In this case, you are creating a client for running service whose WSDL contract is available to you. Using the service's contract, Artix can generate the stub code needed to create a service proxy for your client. It can even generate a sample client for you. Using the generated code, you simply need to finish the consumer's business logic and build the client with the Artix libraries.

Building a service consumer with Artix involves three steps:

1. Importing your service's WSDL contract into Artix.
2. Building the consumer using either C++ or Java.
3. Configuring the Artix runtime to deploy the consumer.

**In this chapter**

This chapter discusses the following topics:

# Importing WSDL into Artix

**Overview**

If you are using the Artix command-line tools you do not need to do anything to get your files into an Artix design environment. However, if you intend to use Artix's Eclipse-based design environment, called Artix Designer, you must create a project in Eclipse and import the contract from which you intend to build the consumer. The advantage of using Artix Designer is that it provides you with an integrated workspace for editing contracts, generating code, editing code, and building runtime artifacts.

**Starting the designer**

On Windows platforms you start the Designer using the **Start** menu shortcut. If you installed Artix using the default settings, the shortcut for starting Artix Designer is under **Start|(All) Programs|IONA| Artix <version>|Artix Designer**. You can also use the `eclipse` executable located under *`InstallDir`*`\artix\`*`version`*`\eclipse`.

On Linux platforms, start the Designer using the supplied `eclipse` executable. It is located in *`InstallDir`*`/artix/`*`version`*`/eclipse`.

When you first start Artix Designer, you are prompted to select a workspace. The workspace is the root folder into which all of your Eclipse projects are stored. You can either accept the recommended default or enter a new workspace.

**Creating a project for your consumer**

Once Artix Designer is started, create an Artix project to build your Web service client. To create a new project and import a contract, do the following:

1. Create a new **Basic Web Services Project** using the **New Project** wizard.
2. Select **File|New|WSDL File** from the Eclipse tool bar to open the **New** wizard.
3. From the **WSDL File** window click **Advanced>>** to open the file selection panel.
4. Select **Link to file in the file system** to make the file selection box available.
5. Click **Browse** to open a file selection dialog.

6. Browse to the location of the contract you want to import and select it.
7. Enter a name for the contract in the **File name** box.
8. Click **Finish** to importing the contract.

The Designer creates a new resource under the selected project that is a shortcut to the original WSDL file on your file system. In addition, it opens the new resource for editing.

# Building the Consumer

**Overview**

Using a WSDL contract, the Artix code generation tools generate most of the code you need to implement a service cconsumer. They generate the stub code needed to instantiate a proxy for the service and generate classes for all of the complex types defined in the contract. In general, coding a consumer using the Artix generated code uses standard C++ or Java APIs.

You will need to use a few Artix-specific APIs to instantiate the Artix bus and to access some of Artix's more advanced features. If you are programming in C++, you will need some Artix-specific code for instantiating your proxy. In addition, some of the types generated by Artix have Artix-specific methods for manipulating them.

The Artix Java interface adheres to the JAX-RPC specification. Thus, working in Java is more standardized. However, Artix does provide some convenience APIs for building client proxies. In addition, using some of the more advanced features require the use of Artix-specific code.

**In this section**

This section discusses the following topics:

# Building a C++ Consumer

**Overview**

Building a consumer using the Artix C++ APIs is a four step process:

1. Generate the stub code from the WSDL contract.
2. Add the code for initializing the Artix bus and a service proxy to the generated code.
3. Add the business logic to your consumer.
4. Build the C++ application.

**Generating code from the command line**

To generate Artix C++ code for a service consumer, do the following:

1. Set up the Artix environment using the following command'

```
ArtixDir/bin/artix_env
```

2. In the directory where the desired WSDL contract is stored, invoke `wsdlgen` as shown below.

```
wsdlgen -cxx client -cxx make -cfg wsdlgenConfig
   wsdl_file.wsdl
```

This will generate C++ classes for all of the complex types defined in your WSDL contract, the C++ stubs for your consumer, a sample `main()`, and a make file for the generated code.

For more information on the `wsdlgen` tool see the Artix WSDLGEN Guide.

**Initializing the consumer**

To create a C++ `main()` for an Artix consumer, do the following:

1. In the same directory as the generated C++ code, create a new C++ file to hold the `main()`.
2. Add the following include statements.

```
#include <it_bus/bus.h>
#include <it_bus/exception.h>
```

3. Add an include statement for the generated header file for your proxy object.

4. Specify that the main is in the same namespace as the generated C++ objects.
5. Specify that the main also uses the `IT_Bus` namespace.
6. If you intend to use any of the complex types defined in the WSDL in your client, add an include statement for the objects the types were generated into.
7. Initialize an instance of the Artix bus using the following command:

```
IT_Bus::init(argc, argv);
```

8. Instantiate an instance of your client using one of the three provided constructors:
   - *portTypeName*`Client()` instantiates a default proxy that uses the first `service` element and `port` element listed in the service's contract. The contract must reside in the same directory from which the application is run.
   - *portTypeName*`Client(`*wsdlPath*`)` instantiates a default proxy using the contract specified by *wsdlPath*.
   - *portTypeName*`Client(`*wsdlPath*`, `*service*`, `*port*`)` instantiates a proxy using the service description specified by the combination of *service* and *port* from the specified contract.

At this point the code for a consumer `main()` will look similar to Example 1.

**Example 1:** *Started C++ Consumer Main*

```
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>

#include "orderWidgetsClient.h"

IT_USING_NAMESPACE_STD

using namespace COM_WIDGETVENDOR_WIDGETORDERFORM;
using namespace IT_Bus;

int main(int argc, char* argv[])
{
  IT_Bus::init(argc, argv);

  orderWidgetsClient client("../widgets.wsdl");

  return 0;
}
```

For more information on instantiating C++ proxies see Developing Artix Applications in C++.

---

**Adding the consumer's business logic**

Once the Artix bus is initialized and the proxy is created, all that remains to create a fully functional service consumer is the consumer's business logic. The code for the consumer's business logic can be any valid C++ code.

For more information on working with Artix generated types see Developing Artix Applications in C++.

# Building a Java Consumer

**Overview**

Building a consumer using the Artix Java APIs is a four step process:

1. Generate the stub code from the WSDL contract.
1. Add the code for initializing the Artix bus and a service proxy to the generated code.
2. Add the business logic to your consumer.
3. Build the Java application.

**Generating code from the command line**

To generate Artix Java code for a service consumer, do the following:

1. Set up the Artix environment using the following command'

```
ArtixDir/bin/artix_env
```

2. In the directory where the desired WSDL contract is stored, invoke `wsdlgen` as shown below.

```
wsdlgen -java client -java ant -cfg wsdlgenConfig
   wsdl_file.wsdl
```

   This will generate the Java classes for all of the complex types defined in your WSDL contract, the Java interface for your consumer, and an ant build target for the generated code.

The generated Java code will be placed in a directory and package structure reflecting the namespaces specified in your contract. For example, if the target namespace of the contract is `widgetVendor.com/widgetOrderForm`, the code for the Java interface will be placed in `com/widgetVendor/widgetOrderForm` and it will be placed in the Java package `com.widgetVendor.widgetOrderForm`. The generated types are placed in a directory and package structure based on the target namespace of the schema in which they are defined.

For more information on the `wsdlgen` tool, see the Artix WSDLGEN Guide.

### Initializing the client

To create a Java `main()` for an Artix consumer do the following:

1. In the same directory as the generated Java interface, create a new Java class to hold the `main()`.
2. Specify that the class is in the same package as the generated Java interface.
3. Add the following import statements to your new class.

```
import java.rmi.RemoteException;
import java.io.*;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
```

4. Add the following Artix specific import statement to your class:

```
import com.iona.jbus.Bus;
```

   `com.iona.jbus.Bus` contains the classes used to implement the Artix bus.
5. If you intend to use any of the complex types defined in the WSDL in your client, add an import statement for the package the types were generated into.
6. Initialize an instance of the Artix bus using the following command:

```
Bus bus = Bus.init(args);
```

7. If you intend to use `anyType` elements, contexts, substitution groups, or SOAP headers, you need to register the generated type factory with the bus.
8. Edit the WSDL path in the generated type factory to contain the correct path to the consumer's contract.
9. Instantiate a new JAX-RPC `ServiceFactory` object using `ServiceFactory.newInstance()`.
10. Instantiate a new JAX-RPC `Service` object using the `ServiceFactory` object's `createService()` method.

`createService()` requires the URL of the service's WSDL contract and the QName of the service's definition in the WSDL contract.

11. Create a JAX-RPC dynamic service proxy using the `Service` object's `getPort()` method.

    `getPort()` requires the QName of the service's port definition in the WSDL contract and the class for the service's interface. The returned port needs to be cast into the appropraite type before it can be used.

12. At the end of the `main()` add the following line to ensure that the bus is properly shut down.

```
bus.shutdown(true);
```

At this point your consumer's `main()` should look similar to Example 2.

**Example 2:** *Started Java Artix main()*

```
package com.widgetvendor.widgetorderform;

import java.rmi.RemoteException;
import java.util.ArrayList;
import java.io.*;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

import com.iona.jbus.Bus;

import com.widgetvendor.types.widgettypes.*;

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

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

**Example 2:** *Started Java Artix main()*

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

    String wsdlPath = "file:../widgets.wsdl";
    wsdlLocation = new URL(wsdlPath);

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

    QName portName = new QName("","widgetOrderPort");
    OrderWidgets impl = (OrderWidgets)service.getPort(portName, OrderWidgets.class);

    bus.shutdown(true);
  }
}
```

For more information about creating dynamic service proxies see Developing Artix Applications with Java.

**Adding the consumer's business logic**

Once the Artix bus is initialized and the service proxy is created, all that remains to create a fully functional service consumer is the consumer's business logic. The code for the business logic can be any valid Java code. The only requirement is that `bus.shutdown()` must be called before the client exits.

For more information on working with Artix generated types see Developing Artix Applications with Java.

**Building the client**

You build the consumer using the standard Java compiler. You need to ensure that the following jars are on your classpath:

- *installdir*\lib\artix\java_runtime\3.0\it_bus-api.jar
- *installdir*\lib\artix\ws_common\3.0\it_wsdl.jar
- *installdir*\lib\artix\ws_common\3.0\it_ws_reflect.jar
- *installdir*\lib\artix\ws_common\3.0\it_ws_reflect_types.jar
- *installdir*\lib\common\ifc\1.1\ifc.jar
- *installdir*\lib\jaxrpc\jaxrpc\1.1\jaxrpc-api.jar

For more information on building a Java application see Developing Artix Applications with Java.

# Configuring Artix to Deploy a Consumer

**Overview**

Once your consumer is built you need to configure the Artix runtime to launch the appropriate services to support the client. You also need to ensure that your consumer is started with the appropriate runtime flags to load the Artix runtime correctly.

**Configuring the Artix runtime**

The Artix runtime configuration is stored in a text file that is stored in the `etc` directory of your Artix installation. This file is typically called `artix.cfg`. It contains settings that control what plug-ins the Artix bus loads, the logging level, and other advanced features.

For basic consumers you do not need to edit `artix.cfg`. However if you need logging functionality, you will need to edit this file. For information on enabling logging see Configuring and Deploying Artix Solutions.

**Loading the runtime configuration**

To ensure that the Artix runtime is loaded do the following:

1. Open a command window.
2. Go to the `bin` directory of your Artix installation.
3. Run `artix_env`.

CHAPTER 2

# Building a Service Provider

*Artix makes building a service provider simple.*

**Overview**

Artix provides the tools to generate the skeleton code needed to quickly develop a service provider. The generated code allows you to use standard APIs, in either C++ or Java, to write the logic to implement your service.

Building a service provider with Artix involves three steps:

1. Defining your service's interface in a WSDL contract.
2. Building the service implementation using either C++ or Java.
3. Configuring the Artix runtime to deploy the service.

**In this chapter**

This chapter discusses the following topics:

# Defining a Service in WSDL

**Overview**

The first step in building a service provider is getting a WSDL contract that defines the service's interface and its physical endpoint details. Often, this contract is provided for the service developer. However, developers do need to build some contracts from scratch or modify existing contracts. This section outlines the steps to build a contract using the Artix Designer. It provides details on adding a few of the more advanced data types using the Designer.

**In this section**

This section discusses the following topics:

# Creating a WSDL Contract

**Overview**

The Artix Designer groups all of the resources used to build an Artix application in a project. Once, you have created an Artix project, you can then build a WSDL contract for your service.

**Starting the designer**

On Windows platforms you start the Designer using the **Start** menu shortcut. If you installed Artix using the default settings the shortcut for starting the Designer is under **Start|(All) Programs|IONA| Artix <version>|Artix Designer**. You can also use the `eclipse` executable located in *`InstallDir`*`\artix\`*`version`*`\eclipse\`.

On Linux platforms, start the Designer using the supplied `eclipse` executable. It is located in *`InstallDir`*`/artix/`*`version`*`/eclipse`.

When you first start Artix Designer, you are prompted to select a workspace. The workspace is the root folder into which all of your Eclipse projects are stored. You can either accept the recommended default or enter a new workspace.

If you do not wish to see this dialog at start-up, place a check in the box labeled **Use this as the default and do not ask again**. Once you select this option, the Designer always opens in the specified workspace. Once the Designer is running, you can change workspaces using **File|Switch Workspace...**.

**Creating a project for your service**

Once the designer is started, you need to create an Artix project to build your service provider. To create an Artix project and a new contract for your service, complete the following steps:

1. Create a new **Basic Web Services Project** using the Eclipse new project wizard.

   **Note:** If asked to switch to the Artix perspective, do so.

2. Select **File|New|WSDL File**.
3. Enter a new for the new contract in the **File Name** field.
4. Click **Finish**.

# Defining the Data Used by the Service

**Overview**

In a WSDL contract the data types used by the service are defined using XML Schema. The type definitions are placed in the contract using the `type` element. The Artix design environment provides wizards for defining many of the XML Schema constructs used to define data type. It also has an XML source editor for entering XML Schema contracts not supported by the designer.

For more information on defining types, see Writing Artix Contracts.

**Adding a simpleType**

To add a simple type to your contract using the designer's wizards do the following:

1. Place the editor into diagram view.
2. RIght-click the **Types** node in the element tree.
3. Select **New Type...** from the pop-up menu.
4. Select at least one resource from the list to act as a source of predefined types.

   All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XMLSchema types. The resources will also be imported to the target resource using WSDL `import` elements.
5. Click **Next>**.
6. Enter a name for the new type.
7. Enter the target namespace for the new type's XML Schema.
8. Under **Kind**, select **simpleType**.
9. Click **Next>**.
10. Chose a base type for the new type from the **Base Type** drop-down list.
11. Add the desired facets for your type.

    **Note:** Artix only supports the `enumeration` facet.
12. Click **Next>**.
13. Review the XML Schema entry for the new type.

14. Click **Finish**.

---

**Adding a complexType**

To add a new `complexType` using Artix designer's diagram view do the following:

1. Right-click the **Types** node to activate the pop-up menu.
2. Select **New Type...** from the pop-up menu.
3. Select at least one resource from the list to act as a source of predefined types.

   All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XMLSchema types. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next>**.
5. Enter a name for the new type.
6. Enter the target namespace for the new type's XML Schema.

   You can either enter a new target namespace manually or, if your resource has multiple schema namespaces defined within it, you can select one of the existing namespaces from the drop-down list.
7. Under **Kind**, select **complexType**.
8. Click **Next>**.
9. Select what type of structure you want to add using the **Group Type** drop-down list.
10. To add elements to the structure click **Add...**.
11. Select the type of the element you want to add to the structure from the **Type:** drop-down list.
12. Enter a name for the new element.
13. Choose if you want the element to be qualified or unqualified using the **Form:** drop-down menu.
14. Specify the min and max occurs attributes for the element.

    The default is to not place the attributes on the element.
15. Click **OK** to save the element to the **Element List** table.

16. If you need to edit an element definition:
    i. Select the element from the **Element List** table.
    ii. Click **Edit...**
    iii. Make any changes you desire.
    iv. Click **OK** to save the changes.
17. Repeat steps **10** through **16** until you have finished adding elements to the structure.
18. Click **Next>**.
19. If you want to add an attribute to the structure click **Add...**.
20. Select a type for the attribute from the **Type:** drop-down menu.
21. Enter a name for the new attribute.
22. Choose if you want the attribute to be qualified or unqualified using the **Form:** drop-down list.
23. If the attribute is going to have a fixed value or a default value enter the value in the **Value:** field.
24. If the attribute is going to be a fixed value, place a check in the **Fixed** check box.
25. If the specified value is a default value, place a check in the **Default** check box.
26. If the attribute must be present in every instance of this type, place a check in the **Required** check box.
27. Click **OK** to save the attribute to the **Attribute List** table.
28. If you need to edit an attribute definition:
    i. Select the attribute from the **Attribute List** table.
    ii. Click **Edit...**
    iii. Make any changes you desire.
    iv. Click **OK** to save the changes.
29. Repeat steps **19** through **28** until you have finished adding attributes to the structure.
30. Click **Next>**.
31. Review the XML Schema for the new type.
32. Click **Finish**.

**Adding an element**

To add an element do the following:

1. Right-click the **Types** node.
2. Select **New Type...** from the pop-up menu.
3. Select at least one resource from the list to act as a source of predefined types.

   All of the predefined types in the selected resources will be made available to you later in the process, as well as the native XML Schema types. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next>**
5. Enter a name for the new type.
6. Enter the target namespace for the new type's XML Schema.

   You can either enter a new target namespace manually or, if your resource has multiple schema namespaces defined within it, you can select one of the existing namespaces from the drop-down list.
7. Under **Kind**, select **Element**.
8. Click **Next>**.
9. If the new element is nilable, place a check in the **Nilable** check box.
10. Select how you intend to define the type of the element.
11. Click **Finish**.

**Editing Types in the designer's source view**

You can also directly edit the XML source for the types. The designer does not provide wizards for a number of the XMLSchema constructs supported by Artix including:

- attribute groups
- lists
- unions

When you save the contract after editing it in source view the designer validates the XML. If it is not valid, the Designer will place errors in the Eclipse **Problems** window.

# Defining the Messages Used by the Service

**Overview**

Once you have defined the data types used by the service, you need to define how that data is organized into the message used by the service to accept requests and send out responses. This is done using `message` elements. The designer provides you with wizards for defining a message and adding the WSDL to your contract. You can also edit the XML source in the designer's source view.

For more information on messages see Writing Artix Contracts.

**Adding a message**

To add a message to your contract do the following:

1. Right-click the **Message** node.
2. Select **New Message...**.
3. Select at least one resource from the list to act as a source of data types.

   All of the types defined in the selected resources, as well as the native XML Schema types, will be made available for you to use in defining messages. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next>**.
5. Enter a name for the message.
6. Click **Next>**.
7. Click **Add...**.
8. Enter a name for the message part.
9. Select a data type from the **Type:** drop-down list.
10. Click **OK** to save the new part to the **Part List** table.
11. If you need to edit a part definition:
    i. Select the part from the **Part List** table.
    ii. Click **Edit...**.
    iii. Make any changes you desire.
    iv. Click **OK** to save the changes.

12. Repeat steps **7** through **11** until you have finished adding parts to the message.
13. Click **Next>**.
14. Review the WSDL for the new message.
15. Click **Finish**.

# Defining the Service's Interface

**Overview**

The operations provided by the service are defined in the contract's `portType` element. The designer provides you with wizards for defining operations and adding the WSDL to your contract. You can also edit the XML source in the designer's source view.

For more information on defining an interface see Writing Artix Contracts.

**Adding a portType**

The add a new interface to your contract do the following:

1. Right-click the **Port Types**.
2. Select **New Port Type...**.
3. Select at least one resource from the list to act as a source of messages.

   All of the messages defined in the selected resources will be made available for you to use in defining the interface's operations. The resources will also be imported to the target resource using WSDL `import` elements.
4. Click **Next>**.
5. Enter a name for the new interface.
6. Click **Next>**.
7. Enter a name for the new operation.
8. Select an operation style from the **Style:** drop-down list.
9. Click **Next>**.
10. Click **Add...**.
11. Select a message type for the new operation message from the **Type:** drop-down list.
12. Select the global message that defines the data passed by this operation message from the **Message:** drop-down list.
13. Enter a name for the operation message.
14. Click **OK** to add the message to the **Operation Messages** table.

15. If you need to edit an operation message:
    i. Select the message from the **Operation Messages** table.
    ii. Click **Edit...**
    iii. Make any changes you desire.
    iv. Click **OK** to save the changes.
16. Repeat steps **10** through **15** until all of the operational messages have been specified.
17. Click **Finish**.

---

**Adding an operation to a port type**

To add a new operation to a port type do the following:

1. Right-click the port type to which you want to add the operation.
2. Select **New Operation...**.
3. Enter a name for the new operation.
4. Select an operation style from the **Style** drop-down list.
5. Click **Next>**.
6. Click **Add...**.
7. Select a message type for the new operation message from the **Type:** drop-down list.
8. Select the message that defines the data passed by this operation message from the **Message:** drop-down list.
9. Enter a name for the operation message.
10. Click **OK** to add the message to the **Operation Messages** table.
11. Repeat steps **6** through **10** until all of the operation's messages have been specified.
12. Click **Finish**.

# Defining the Payload Format

**Overview**

After defining the logical structure of your service and the data it uses, you must now define the concrete representation of the data that will be passed on the wire. This is done in the `binding` element of a contract. Web service providers typically use SOAP as the payload format.

For more information on defining a payload format see Writing Artix Contracts.

**Adding a SOAP binding using the designer**

To add a SOAP 1.1 binding do the following:

1. Right click the **Bindings** icon.
2. Select **New Binding...**.
3. Select at least one contract from the list to act as a source for interfaces.

   All of the interfaces in the selected contracts will be made available to you later. The contracts will also be imported to the target resource using WSDL `import` elements.
4. Click **Next>**.
5. Select **SOAP 1.1**.
6. Click **Next>**.
7. Select the interface which is being mapped to this SOAP binding from the **Port Type** drop-down list.
8. Enter a name for the binding.
9. Select a style for the SOAP elements from the **Style** drop-down list.
10. Select a value for the SOAP `use` attribute from the **Use** drop-down list.
11. Edit the attributes of the individual portions of the SOAP binding.
12. Click **Next>**.
13. Review the binding.
14. Click **Finish**.

# Defining the Service Provider's Endpoint Information

**Overview**

The final piece of information needed to fully define a service provider is the transport the service uses and its network address. This information is provided in the contract's `service` element. Web service providers typically use HTTP as their transport.

For more information on defining an endpoint see Writing Artix Contracts.

**Adding an HTTP port using the designer**

To add an HTTP port to your contract from the designer's diagram view do the following:

1. Right click the **Services** icon.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.

   All of the bindings defined in the selected contracts will be made available for you to use in defining a service. The contracts will be imported to contract using WSDL `import` elements.
4. Click **Next>**.
5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.
8. Select a binding to be exposed by this port from the **Binding** drop-down list.
9. Click **Next>**.
10. Select **SOAP/HTTP** from the **Transport Type** drop-down list.
11. Enter the HTTP address for the port in the **location** field of the **Address** table.
12. Enter values for any of the optional configuration settings you desire.
13. Click **Next>**.
14. Review the WSDL for the service definition.
15. Click **Finish**.

# Building the Service Provider

**Overview**

Using a WSDL contract, the Artix code generation tools generate most of the code you need to implement a service provider. They generate the skeleton code needed to deploy your service provider into an Artix container. In addition, they generate classes for all of the complex types defined in the contract. In general, coding a service provider using the Artix generated code uses standard C++ or Java APIs.

You will need to use a few Artix specific APIs to initialize the runtime and to access some of Artix's more advanced features. In addition, some of the generated types have Artix specific methods for manipulating them.

The Artix Java interface adheres to the JAX-RPC specification. Working in Java is, therefore, more standardized.

**In this section**

This section discusses the following topics:

# Building a Service Provider in C++

**Overview**

Building a service provider using the Artix C++ APIs is a four step process:

1. Generate the skeleton code from the WSDL contract.
2. Add the code required to instantiate the service provider plug-in and register a servant with the runtime.
3. Implement the service provider's business logic.
4. Build the service provider plug-in.

**Generating code**

To generate C++ code for a service provider do the following:

1. Set up the Artix environment using the following command.

   ```
   ArtixDir/bin/artix_env
   ```

2. In the directory where the desired WSDL contract is stored, invoke `wsdlgen` as shown below.

   ```
   wsdlgen  -D portType=portType -cxx plugin -cxx make -cfg
      wsdlgenConfig wsdl_file.wsdl
   ```

   This will generate C++ classes for all of the complex types defined in your WSDL contract, the C++ skeletons for your service provider, a make file for the generated code, and the plug-in classes needed to load the service provider into the Artix container.

For more information on the `wsdlgen` tool see the Artix WSDLGEN Guide.

**Editing the plug-in classes**

The plug-in classes generated by the code generators are responsible for creating instances of your service provider's servant. There are two classes generated:

- *PortTypeName*`ServantPlugIn` - the main class of your plug-in. It has one method, `bus_init()`, that is responsible for creating instances of your service provider's implementation class and registering them as servants with the runtime. It also has a method, `bus_shutdown()`, that is responsible for cleaning up any resources used by the plug-in when it exits.

- *PortTypeName*`ServantPlugInFactory` - a factory class used by the runtime to create instances of your service provider's plug-in.

For basic service providers you will not need to edit the generated classes. However, if you want to initiate any resources or use the plug-in to host multiple service providers, you can edit the `bus_init()` and `bus_shutdown()` methods. For more information see Developing Artix Applications with C++.

---

**Implementing the service**

The code generator generates a shell implementation class named for the `portType` element defining the interface in the WSDL contract. The class name is generated by appending `Impl` on the value of the `name` attribute in the `portType` element that defines the service's interface. For example, if the vale of the `portType` element's name attribute is `orderWidgets`, the generated implementation class would be `orderWidgetsImpl`.

The generated class will have one method for each operation defined in the WSDL contract. The parameter list for each method will be generated as described in Developing Artix Applications in C++.

You can implement the methods using standard C++ methods. In addition, Artix provides a number of proprietary APIs for using advanced functionality and working with the generated data types.

# Building a Service Provider in Java

## Overview

Building a service provider using the Java APIs is a four step process:

1. Generate the skeleton code from the WSDL contract.
2. Add the code required to instantiate the service provider plug-in and register a servant with the bus.
3. Implement the service provider's business logic.
4. Build the service provider plug-in.

## Generating code

To generate Java code for a service provider do the following:

1. Set up the Artix environment using the following command'

   ```
   ArtixDir/bin/artix_env
   ```

2. In the directory where the desired WSDL contract is stored, invoke `wsdlgen` as shown below.

   ```
   wsdlgen -D portType=portType -java plugin -java ant -cfg
      wsdlgenConfig wsdl_file.wsdl
   ```

   This will generate the Java classes for all of the complex types defined in your WSDL contract, the Java interface for your service, a shell object for your implementation object, the plug-in classes needed to load the service provider into the Artix container, and an ant build target for the generated code.

The generated Java code will be placed in a directory and package structure reflecting the namespaces specified in your contract. For example if the target namespace of the contract is `widgetVendor.com/widgetOrderForm`, the code for the Java interface will be placed in `com/widgetVendor/widgetOrderForm` and it will be placed in the Java package `com.widgetVendor.widgetOrderForm`. The generated types are placed in a directory and package structure based on the target namespace of the schema in which they are defined.

For more information on the `wsdlgen` tool, see the Artix WSDLGEN Guide.

For more information on how Java package names are generated, see Developing Artix Applications in Java.

### Editing the plug-in classes

The plug-in classes generated by the code generators are responsible for creating instances of your service provider's servant. There are two classes generated:

- *PortTypeName*`ServicePlugIn` - the main class of your plug-in. It has one method, `busInit()`, that is responsible for creating instances of your service provider's implementation class and registering them as servants with the bus. It also has a method, `busShutdown()`, that is responsible for cleaning up any resources used by the plug-in when it exits.
- *PortTypeName*`ServicePlugInFactory` - a factory class used by the runtime to create instances of your service's plug-in.

For basic service providers you will only need to make one change to the generated plug-in class. Do the following:

1. Open the source file for your service provider's plug-in class.
2. Locate the implementation of the `busInit()` method.
3. Edit the line of code for instantiating a servant so that the value of the second parameter to points to the location of your contract.

```
Servant servant = new SingleInstanceServant(bank,
                                            "./bank.wsdl",
                                            bus);
```

However, if you want to initiate any resources, use the plug-in to host multiple service providers, or use a different threading model you can edit the `busInit()` and `busShutdown()` methods. For more information see Developing Artix Applications with Java.

### Implementing the service

The code generator generates a shell implementation class named for the `portType` defining the interface in the WSDL contract. The class name is generated by appending `Impl` on the value of the name attribute in the `portType` that defines the interface. For example, if the vale of the `portType` element's name attribute is `orderWidgets`, the generated implementation class would be `OrderWidgetsImpl`.

The generated class will have one method for each operation defined in the WSDL contract. The parameter list for each method will be generated as described in Developing Artix Applications in Java. If the operation has a return value, the generated method will contain a return statement that returns a dummy value. If the operation has no return value, it is left empty.

You can implement the methods using standard Java methods. In addition, Artix supports the use of JAX-RPC `MessageContext` objects and provides a number of proprietary APIs for using advanced functionality.

For more information on working with generated types see Developing Aritx Applications with Java.

---

**Building the service**

You build the service provider using the standard Java compiler. You need to ensure that the following jars are on your classpath:

*installdir*\lib\artix\java_runtime\3.0\it_bus-api.jar

*installdir*\lib\artix\ws_common\3.0\it_wsdl.jar

*installdir*\lib\artix\ws_common\3.0\it_ws_reflect.jar

*installdir*\lib\artix\ws_common\3.0\it_ws_reflect_types.jar

*installdir*\lib\common\ifc\1.1\ifc.jar

*installdir*\lib\jaxrpc\jaxrpc\1.1\jaxrpc-api.jar

For more information on building a Java Artix application see Developing Aritx Applications with Java.

# Configuring Artix to Deploy the Service Provider

**Overview**

Once your service provider is built you need to configure Artix to launch the appropriate services to support the service provider. You also need to ensure that your service provider is started with the appropriate runtime flags to load the Artix runtime correctly.

**Configuring the Artix runtime**

The Artix runtime configuration is stored in a text file that is stored in the `etc` directory of your Artix installation. This file is typically called `artix.cfg`. It contains settings that control what plug-ins Artix loads, the logging level, and other advanced features.

For basic service providers you do not need to edit `artix.cfg`. However, if you need logging functionality, you will need to edit this file. For information on enabling logging see Configuring and Deploying Artix Solutions.

**Loading the runtime configuration**

To ensure that the Artix runtime is loaded do the following:

1. Open a command window.
2. Go to the `bin` directory of your Artix installation.
3. Run `artix_env`.

**Starting the service provider**

Artix service providers are launched inside the Artix container. The Artix container is a lightweight framework for hosting and managing service providers. You can deploy your service provider into a container in one of two ways:

**Deploying a service provider into a new container**

To deploy your service provider in a new Artix container you use the `it_container` command as shown in Example 3.

**Example 3:** *Deploying a Service Provider in a New Container*

```
it_container -deploy widgets.xml
```

The `it_container` command's `-deploy` flag requires the name of the deployment descriptor generated for your service provider. It is named *`PortTypeName`*`.xml`.

### Deploying a service provider into a running container

To deploy your service provider in a new Artix container you use the `it_container_admin` command as shown in Example 3.

**Example 4:** *Deploying a Service Provider in a Running Container*

```
it_container_admin -deploy -file widgets.xml
```

The `it_container_admin` command's `-file` flag requires the name of the deployment descriptor generated for your service provider. It is named *`PortTypeName`*`.xml`.

CHAPTER 3

# Service Enabling Backend Services

*Artix allows you to quickly expose backend services, that use a variety of middlewares, as services. Often, you do not need to write any new code.*

**Overview**

Artix provides the tools to define a backend service in WSDL and then expose it as a service. In most cases, there is no additional coding needed. Artix does all of the work.

Exposing a backend service as a service with Artix involves the following steps:

1. Creating a WSDL description of your service.
2. Defining an endpoint for your service.
3. Configuring the Artix runtime to deploy an Artix router to expose your service.

**In this chapter**

This chapter discusses the following topics:

# Describing a Service in WSDL

**Overview**

The first step in service enabling an existing system with Artix is to describe the system in an Artix contract. Artix contracts are WSDL documents that describe a service's operations on two levels. The first is the abstract level where the data and messages exchanged by the service are described using XML Schema. The second level is the concrete level where the messages and data are bound to a concrete format and the service's communication details are defined.

For detailed discussion about how to write WSDL documents see Writing Artix Contracts.

**In this section**

This section discusses the following topics:

# Starting with CORBA IDL

**Overview**

If your backend service is implemented using CORBA, it probably has an IDL definition that is accessible. Artix can import a CORBA IDL file and generate a WSDL document representing the CORBA service. The generated WSDL document will have all of the types, messages, and interfaces defined by the IDL represented. In addition, it will have a CORBA payload format definitions and a section defining how the XML Schema types are mapped to the original CORBA types.

When Artix generates a WSDL document from CORBA IDL, it also adds a definition for a CORBA endpoint to the contract. This endpoint definition is incomplete. To complete the definition of the service's endpoint see "Defining the Service's Endpoint Information" on page 51.

You can generate WSDL from IDL using either the Artix designer or the `idltowsdl` command line tool.

**Using Artix Designer**

To use an IDL file as the basis for a contract:

1. Create a project for your switch.
2. From the **File** menu, select **New|WSDL from IDL**.
3. Select the folder where you want to store the WSDL file.
4. Type the name of contract in the **File name** field.
5. Click **Next>**.
6. Enter the pathname for the IDL file.
7. Under **Object Reference** select the method you will use to specify the CORBA service's object reference.

**Using a Naming Service**

i. Enter the corbaloc for the naming service that holds the reference to the CORBA service you wish to access and click **Browse...** to open the naming service browser.

ii. Select the service you wish to expose from the browser.

iii. The radio button under **Object Reference** will change to **IOR or CORBA URL** and the service's IOR will be placed in the text box.

#### Using an IOR or CORBA URL

Enter either a stringified CORBA IOR, a corbaname URL, or a corbaloc URL into the text box.

#### Using an IOR from a File

Use the **Browse** button to locate the name of a file that contains the stringified IOR of the CORBA service or enter the name of the file in the text box.

8. Click **Finish**.

---

### Using idltowsdl

To create a WSDL file from CORBA IDL do the following:

1. From the `bin` directory of your Artix installation, source the Artix environment by running `artix_env`.
2. Find the IDL file for the service you want to expose.
3. Run `idltowsdl` as shown below.

```
idltowsdl idlfile
```

For more information on the `idltowsdl` tool see Artix for CORBA.

# Starting with a COBOL Copybook

**Overview**

Many backend services are written in COBOL and the data used by these systems is defined in a COBOL copybook. Artix can import a COBOL copybook and generate a WSDL document defining the service. The resulting WSDL document will have definitions for the types, messages, and an interface for a service based on the COBOL copybook. In addition, it will contain a payload format definition that enables the service to communicate using fixed format data.

If your service uses another payload format you can change the payload format definition. For information on defining new payload formats see Writing Artix Contracts.

Artix does not add any endpoint information when generating WSDL from a COBOL copybook. To add endpoint information see "Defining the Service's Endpoint Information" on page 51.

You can generate WSDL from a COBOL copybook using either the Artix Designer or the `coboltowsdl` command line tool.

**Using Artix designer**

To add a contract containing a fixed binding from a COBOL copybook:

1. From the **File** menu, select **New|WSDL From Dataset**.
2. Select the folder where you want to store the WSDL file.
3. Enter a name for the contract.
4. Click **Next>**.
5. Select **Fixed from a COBOL Copybook**.
6. Click **Next>**.
7. Enter the name for the generated `binding` element in the **Binding Name** field.
8. Enter the name for the generated `portType` element in the **Port Type Name** field.
9. Enter the namesapce for the generated contract's target namespace in the **Target Namespace** field.
10. Enter the namspace you wish to use as the target namespace for the generated contract's `schema` element in the **Schema Namespace** field.

11. Place a check in the **Create message parts with elements** check box if you want to generate a contract where the message parts are defined using `element` elements.
12. Select the justification value to use in the generated fixed binding from the **Justification** drop-down list.
13. Enter the character encoding you wish to use for the generated fixed binding in the **Encoding** field.
14. Enter the character to use for padding on the wire in the **Padding** field.
15. Click **Next>**.
16. Click **Add** to add a new operation to the table.
17. Select the new operation from the table.
18. Click **Edit** to bring up the operation editing table.
19. Enter a name for the operation.
20. Select what type of operation you want to define from the **Style** drop-down list.
21. Enter a discriminator string for the operation in the **Discriminator** field.
22. Select one of the messages from tree on the left to bring up the message editing table.
23. Enter a name for the message.
24. Select the type of message from the **Type** drop-down list.
25. Enter the desired attributes for the message.
26. Click **Import COBOL Copybook** to bring up a file browser.
27. Select the copybook that defines the message from the file browser.
28. Repeat steps **22** through **27** for each message in the operation.
29. Repeat steps **16** through **28** for each operation in the service you are defining.
30. Click **Finish**.

---

**Using coboltowsdl**

To create a WSDL document from a COBOL copy book using `coboltowsdl` do the following:

1. From the `bin` directory of your Artix installation, source the Artix environment by running `artix_env`.
2. Find the COBOL copybook file for the service you want to expose.

3. Run `coboltowsdl` as shown below.

```
coboltowsdl -b bindingName -op operationName
            -im inputName:inputCopybook
           [-om outputName:outputCopybook]
```

- *`bindingName`* is the name of the generated fixed data binding in the resulting WSDL document.
- *`operationName`* is the name of the generated operation in the resulting WSDL document.
- *`inputName`* is the name of the generated input message in the resulting WSDL document.
- *`inputCopybook`* is the name of the COBOL copybook that contains the data definitions for the input message.
- *`outputName`* is the name of the generated output message in the resulting WSDL document.
- *`outputCopybook`* is the name of the COBOL copybook that contains the data definitions for the output message.

For more information on the `coboltowsdl` tool see Writing Artix Contracts.

# Starting with a Java Class

**Overview**

Artix can import compiled Java classes and generate WSDL documents representing the types, messages, and interfaces used by the Java methods defined in the class. The mapping used to generate WSDL from Java is based on the JAX-RPC 1.1 specification.

**Using Artix designer**

The Artix designer can only create contracts from class files that are part of an Eclipse Java project in the same workspace as your Artix project. To create a contract from a Java class using the Artix designer do the following:

1. From the **File** menu, select **New|WSDL From Java**.
2. Select the folder where you want to store the WSDL file.
3. Enter a name for the contract.
4. Click **Next>**.
5. Click **Browse...** to open the Java class browser.
6. Enter a search string to locate the class you wish to import in the **Select a class to use** field.
7. Select the desired class from the list of types in the **Matching types:** list.
8. Click **OK**.
9. If you do not want to use the default values in the generated contract, place a check in the **WSDL settings** check box if you do not want Artix to use default values in the generated contract and specify the desired values.
10. Click **Finish**.

**Using javatowsdl**

To generate WDSL from a compiled Java class do the following:

1. From the `bin` directory of your Artix installation, source the Artix environment by running `artix_env`.
2. Find the Java class file for the service you want to expose.
3. Run `javatowsdl` as shown below.

```
javatowsdl className
```

# Defining the Service's Endpoint Information

**Overview**

Once you have your service defined logically and have added the proper message format binding, you need to define the physical transport details that expose the service as an endpoint. Depending on the transport used by your service, the endpoint details can be as simple as specifying an HTTP port or as complicated as specifying the JMS topics used to send and receive messages. Artix uses a number of proprietary WSDL extensions to define endpoint details for a service.

To define an endpoint you need to create two WSDL elements:

1. A `service` element that contains a list of endpoints.
2. A `port` element that contains the details for a specific endpoint.

**Defining a CORBA endpoint**

To add a CORBA endpoint to your contract do the following:

1. Right click the **Services** node to activate the pop-up menu.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.
4. Click **Next>**.
5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.
8. Select a CORBA binding to be exposed by this port.
9. Click **Next>**.
10. Enter a valid CORBA address in the **location** field of the **Address** table.
11. Set any desired POA policies in the **Policy** table.
12. Click **Finish**.

**Defining a WebsphereMQ endpoint**

To add a WebsphereMQ endpoint to your contract do the following:

1. Alt-click the **Services** node to activate the pop-up menu.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.
4. Click **Next>**.

5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.
8. Select a binding to be exposed by this port.
9. Click **Next>**.
10. Select **MQ** from the **Transport Type** drop-down list.
11. If you are adding a port for an MQ client, enter valid names in the **QueueName** field and the **QueueManager** field of the **Client** table.
12. If you are adding a port for an MQ client that will be getting responses from its server, enter valid names in the **ReplyQueueName** field and the **ReplyQueueManager** field of the **Client** table.
13. If you are adding a port for an MQ server, enter valid names in the **QueueName** field and the **QueueManager** field of the **Server** table.
14. Edit any of the remaining optional attributes.
15. Click **Finish**.

---

**Defining a Tuxedo endpoint**

To add a Tuxedo endpoint to your contract do the following:

1. Alt-click the **Services** node to activate the pop-up menu.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.
4. Click **Next>**.
5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.
8. Select a binding to be exposed by this port.
9. Click **Next>**.
10. Select **Tuxedo** from the **Transport Type** drop-down list.
11. Click the **Add** button under the **Services** table to add a Tuxedo service to the table.
12. Click in the **Attribute** column to edit the service's name.
13. With the service selected in the **Service** table, click the **Add** button under the **Operations** table.

14. Select one of the operations from the window that pops up.
15. Click **OK** to return to editing the transport properties.
16. Repeat steps **13** through **15** until you have added all of the desired operations for the service.
17. Repeat steps **11** through **16** until you have added all of the desired Tuxedo services to the port.
18. Click **Finish**.

---

**Defining a Tibco endpoint**

To add a Tibco/RV endpoint to your contract do the following:

1. Alt-click the **Services** node to activate the pop-up menu.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.
4. Click **Next>**.
5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.
8. Select a binding to be exposed by this port.
9. Click **Next>**.
10. Select **Tib/Rv** from the **Transport Type** drop-down list.
11. Specify the name of the subject to which the server listens in the **serverSubject** field.
12. Set any desired optional attributes.
13. Click **Finish**.

---

**Defining a JMS endpoint**

To add a JMS endpoint to your contract do the following:

1. Alt-click the **Services** node to activate the pop-up menu.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.
4. Click **Next>**.
5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.

8. Select a binding to be exposed by this port.
9. Click **Next>**.
10. Select **JMS** from the **Transport Type** drop-down list.
11. Set the required port properties.
12. Click **Finish**.

# Defining a SOAP/HTTP Endpoint

**Overview**

In order to expose a service as a Web service you need to create an endpoint that exposes your service using SOAP as its message format and HTTP as its network transport. You also need to create a logical connection, or route, between your service's native endpoint and the SOAP/HTTP endpoint. Artix uses this route to determine how to translate the messages between your service's native format and SOAP.

Adding a SOAP/HTTP endpoint and connecting it your service's native interface is a three step process:

1. Create a SOAP binding for your service.
2. Define an HTTP endpoint to expose your service using SOAP.
3. Define the route that connects your service's native endpoint to the new SOAP/HTTP endpoint.

**Defining a SOAP binding for your service**

To add a SOAP binding from the designer's diagram view do the following:

1. Alt-click the **Bindings** node to activate the pop-up window.
2. Select **New Binding...**.
3. Select at least one contract from the list to act as a source for interfaces.
4. Click **Next>**.
5. Select **SOAP 1.1**.
6. Click **Next>**.
7. Select the interface which is being mapped to this SOAP binding.
8. Enter a name for the binding.
9. Select a style for the SOAP elements from the **Style** drop-down list.
10. Select a value for the SOAP `use` attribute from the **Use** drop-down list.
11. Click **Finish**.

### Using the command line

In addition to the designer, Artix provides a command line tool, `wsdltosoap`, that will create a SOAP payload binding for a specified port type in a WSDL document. You execute `wsdltosoap` as shown below.

```
wsdltosoap -i portType -n namespace wsdl_file
```

*`portType`* specifies the name of the port type for which to generate the SOAP binding. *`namespace`* specifies the namespace used for the generated SOAP binding.

For more information about using the `wsdltosoap` tool see Writing Artix Contracts.

### Defining an HTTP endpoint for your service

To add HTTP endpoint to your contract do the following:

1. Right-click the **Services** node to activate the pop-up menu.
2. Select **New Service...**.
3. Select at least one resource from the list to act as a source of bindings.
4. Click **Next>**.
5. Enter a name for the new service.
6. Click **Next>**.
7. Enter a name for the new port.
8. Select a binding to be exposed by this port.
9. Click **Next>**.
10. Select **SOAP?HTTP** from the **Transport Type** drop-down list.
11. Enter the HTTP address for the port in the **location** field of the **Address** table.
12. Enter values for any of the optional configuration settings you desire.
13. Click **Finish**.

### Defining the connection between the native endpoint and the Web service endpoint

Once you have the SOAP binding and HTTP endpoint for your service defined, you need to create a logical pathway, or route, between your service's native endpoint and the Web service endpoint. To add this route do the following:

1. Right-click the **Routes** node to activate the pop-up menu.
2. Select **New Route...**.

3. Select at least one contract from the list to act as a source of services between which to route.
4. Click **Next>**.
5. Enter a name for the new route.
6. Select the interface that is bound to the service that will be the source endpoint for the route from the **Port Type** drop-down list.
7. Select one service from the **Source Endpoint** table to be the source endpoint for the route.
8. Select **Single** from **Destination Preferences**.
9. Select the endpoint that you want to be destinations from the **Destination Endpoints** table.
10. Click **Next>**.
11. Select at least one operation to use in the route.
12. Click **Next>**.
13. Define an attribute routing rule.
14. Click **Add**.
15. Repeat steps **13** and **14** until you have added all of the desired attribute routing rules.
16. Click **Finish**.

For more information about defining a route see the Artix Router Guide.

# Configuring and Deploying an Artix Router

**Overview**

Once you have created an Artix contract defining your service and added a SOAP/HTTP endpoint to it, you need to deploy the Artix process that will enable your backend service to be seen as a Web service to other applications. This Artix process uses the logical pathway, or route, you defined to translate the SOAP/HTTP message sent to your service and forward them to the actual backend service in its native format.

Configuring and deploying this Artix router is a simple procedure that involves editing a simple text file. It is done in four steps:

1. Create a configuration domain for your process.
2. Specify which plug-ins Artix needs to load.
3. Configure the routing plug-in to locate your Artix contract.
4. Deploy the Artix container with the new configuration information.

**Creating a configuration domain**

To create a new configuration domain for your new Web service do the following:

1. In the folder *install_dir*/artix/*version*/etc/domians, create a new file called *domain_name*.cfg. *domain_name* can be any valid file name as long as it is unique among the other files in the same folder.
2. Open the file in any text editor.
3. Add the following line to import the default configuration settings.

```
include "artix.cfg";
```

4. Create scope in your new domain by adding the following to the file. *scope_name* can be any string value that does not contain spaces or back slashes(\).

```
scope_name
{
}
```

### Specifying the plug-ins to load

The Artix service that handles routing is implemented as a plug-in, so you need to configure Artix to load it when it starts up. In addition, you may want to load one of the logging plug-ins to provide logging information.

To specify the plug-ins loaded by Artix at start-up do the following:

1. Open your new configuration file in any text editor.
2. Inside of the scope you created for the process add the following line.

```
orb_plugins=[]
```

3. Inside of the brackets of the `orb_plugins` line add `"routing"`.
4. If you want to load a logging plug-in, add `"xml_log_stream"` to the beginning of the `orb_plugins` list and separate it from the routing entry by a comma.

### Specifying the location of the contract

The last piece of configuration needed for an Artix router is to tell the routing plug-in where its contract is located. To specify the location of the contract do the following:

1. Open your new configuration in any text editor.
2. Inside the scope you created for the process add the following line. *`wsdl_location`* is the location of the contract containing the routing rules for your Web service enabled service.

```
plugins:routing:wsdl_url="wsdl_location";
```

### Deploying the Artix container

The easiest way to run an Artix router is to launch it inside of the Artix container. To launch the Artix container with a specific configuration use the following command.

```
start it_container -ORBname scope_name -ORBdomain_name domain_name
```

*`scope_name`* specifies the name of the scope you created in your conguration domain. *`domain_name`* specifies the name of the domain you created.

**Example**

Example 5 shows an example of a configuration domain, `widgets.cfg`, that contains the information to launch an Artix router.

**Example 5:** *Sample Routing Configuration*

```
include "artix.cfg";

corba_ws
{
  orb_plugins = ["xmlfile_log_stream", "routing"];

  plugins:routing:wsdl_url="widgets.wsdl";
};
```

To launch the Artix container using the configuration shown in Example 5 you would use the following command.

```
start it_container -ORBname corba_ws -ORBdomain_name widgets
```

CHAPTER 4

# Using Multiple Versions of Xerces

*The Artix classloader firewall can be used to allow an application to use multiple versions of the Xerces XML parser.*

**Overview**

There are occasions when versions of Java libraries used by an Artix application may conflict with the versions of libraries used by the Artix runtime.

The two libraries which are most likely to cause a problem are the Xerces and Log4j libraries. If, for instance, a user wants to use a Xerces version that conflicts with the version used by the Artix runtime, problems may result.

In this situation, an application can make use of an Artix Firewall Classloader that allows different versions of key libraries to be loaded into a JVM without interfering with the functionality of Artix.

**Caution**

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.

```
java:generated_type_handler:disabled=true
```

- Set a `Bus` property.

```
hashtable.put("generated_type_handler.disabled", "true");
Bus bus = Bus.init(args, hashtable);
```

---

**Procedure**

To configure your application to use its own version of the Xerces XML parser using the Artix classloader firewall do the following:

1. Create a file called `artix_ce.xml` and place it on your application's classpath.
2. Enter the Artix classloader firewall configuration 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">
```

3. Enter the parent element of the classloader firewall configuration document.

```
<ce:classloader-environment  xmlns:ce="http://www.iona.com/ns/classloader-environment"
                             loglevel="info">
...
</ce:classloader-environment>
```

4. Enter the `ce:environment` element for the firewall.
5. Set the value of the `ce:environment` element's `name` attribute to `artix_ce`.

```
...
<ce:environment name="artix_ce">
  ...
</ce:environment>
...
```

6. Add the set of filters shown in Example 6 to your classloader firewall's configuration.

**Example 6:** *Filters for Blocking the Artix Xerces XML Parser*

```
...
<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>
...
```

7. Add any other filters you desire to the `ce:firewall` element.
8. Add a `ce:loader` element to the classloader firewall configuration.
9. Add a `ce:location` child to the `ce:loader` element that loads the `java_runtime-rt` jar file.

```
...
<ce:loader>
  <ce:location>usr/lib/artix/java_runtime/4.1/java_runtime-rt.jar</ce:location>
</ce:loader>
...
```

**Example**

Example 7 shows a classloader firewall configuration that will block the Xerces XML parser used by the Artix runtime from your application.

**Example 7:** *Classloader Firewall Configuration for Blocking Xerces*

```
<?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:environment name="artix_ce">
    <ce:firewall>
      <ce:filter type="discover" discover-source="jre"/>
      <ce:filter type="negative-pattern">com.iona.jbus.jms.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.runtime.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.types.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.jaxrpc.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.ntv.</ce:filter>
      <ce:filter type="negative-pattern">com.iona.jbus.util.</ce:filter>
      <ce:filter type="pattern">com.iona.jbus.</ce:filter>
      <ce:filter type="pattern">com.iona.jbus.servants.</ce:filter>
      <ce:filter type="pattern">com.iona.webservices.reflect.types.</ce:filter>
      <ce:filter type="pattern">com.iona.schemas.references</ce:filter>
      <ce:filter type="pattern">javax.xml.rpc.</ce:filter>
      <ce:filter type="pattern">javax.xml.namespace.QName</ce:filter>
    </ce:firewall>
    <ce:loader>
      <ce:location>usr/lib/artix/java_runtime/4.1/java_runtime-rt.jar</ce:location>
    </ce:loader>
  </ce:environment>
</ce:classloader-environment>
```

CHAPTER 5

# Using Artix with .NET

*Artix easily integrates with .NET applications.*

**Overview**

Microsoft .NET is one of the more popular methods for developing Web services. Artix clients and Artix servers that use SOAP over HTTP can directly access .NET applications. This also means that you can use Artix to bridge between a .NET application and any backend service that uses a transport supported by Artix. In addition, .NET applications can interact with the Artix locator and the Artix session manager.

**In this chapter**

This chapter discusses the following topics:

# Building a .NET Client for an Artix Service

**Overview**

If you have a service that was developed using Artix and you want to access it using clients written in .NET, the process is straightforward if the Artix service exposes a SOAP/HTTP endpoint. Visual Studio can import the service's contract from either a running instance of the service or from a local copy of the service's contract. Once the contract is imported, you can develop the client using C#.

If your .NET clients need to communicate with services that use protocols other than SOAP/HTTP, Artix can be used as a bridge. This is shown in "Using Artix to Bridge from a Backend Service to .NET" on page 71. If you want your clients to directly interact with services that use protocols other than SOAP/HTTP, you need to use the Artix Connect software.

**Contract limitations**

.NET can read most of the proprietary extensions Artix uses for transport configuration. However, .NET does not recognize, or make use of, any of the Artix contract elements under the `http-conf` namespace. This means that any client-side transport configuration stored in the contract will not be used. If it is required, you must be sure to set it up for your client using the appropriate .NET methods.

**Procedure**

To create a .NET client for a service developed using Artix, do the following:

1. Create a new empty C# project in Visual Studio.
2. Add a new Web Reference to your project to bring up the Web Reference browser shown in Figure 1.

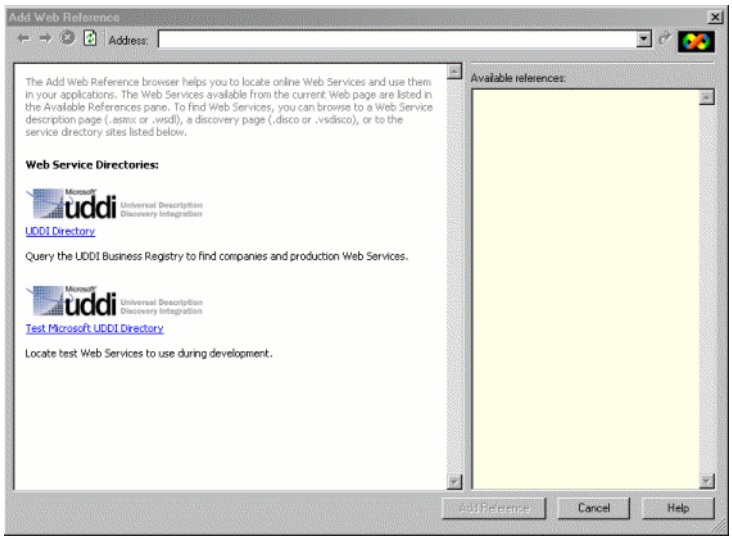


**Figure 1:** *Browsing For a Web Reference in Visual Studio*

3. In the **Address** field of the browser, type in the location of the service's contract.
   - If the service is deployed and has the Artix WSDL publishing plug-in configured, you can enter the address of the running service to get the service's contract.
   - If you have a copy of the service's contract stored on an accessible file system, you can enter the full path name of the contract.
4. Once the contract is loaded, click the **Add Reference** button to add the service to your project.
5. Add a new C# class to your project.

6. When developing your client, instantiate a proxy using the name of the service definition in the imported contract.

When Visual Studio builds your client, it automatically generates the proxy code needed to invoke on the running Artix service.

---

**Example**

Example 8 shows the .NET code for a client that works with the Artix `HelloWorldService`. In this case, the contract was imported into the same namespace as the client.

**Example 8:** *.NET Client for Working with HelloWorldService*

```
using System;

namespace HelloWorldClient
{
  class Client
  {
    static void Main(string[] args)
    {
      HelloWorldService service = new HelloWorldService();

      string str_out, str_in;

      str_out = service.sayHi();
      Console.WriteLine("sayHi method returned: " + str_out);

      str_in = "Early Adopter";
      str_out = service.greetMe(str_in);
      Console.WriteLine("greetMe method returned: " + str_out);
    }
  }
}
```

# Building an Artix Client for a .NET Service

**Overview**

.NET services publish standard WSDL contracts that Artix can use to generate a service proxy. Once you have the WSDL contract defining the .NET service, writing a client using Artix follows the same pattern as writing any other Artix client.

**Procedure**

To build an Artix client for a .NET service, do the following:

1. Get the WSDL for the .NET service by entering its URL appended with `?wsdl` into a Web browser.

   For example, if you are running the .NET `HelloWorld` service shipped as part of the .NET integration demo you would enter

   ```
   http://localhost/HelloWorld_doclit/HelloWorldService.asmx?wsd
   ```

`1` into your browser to display the service's WSDL. The result is shown in Figure 2.

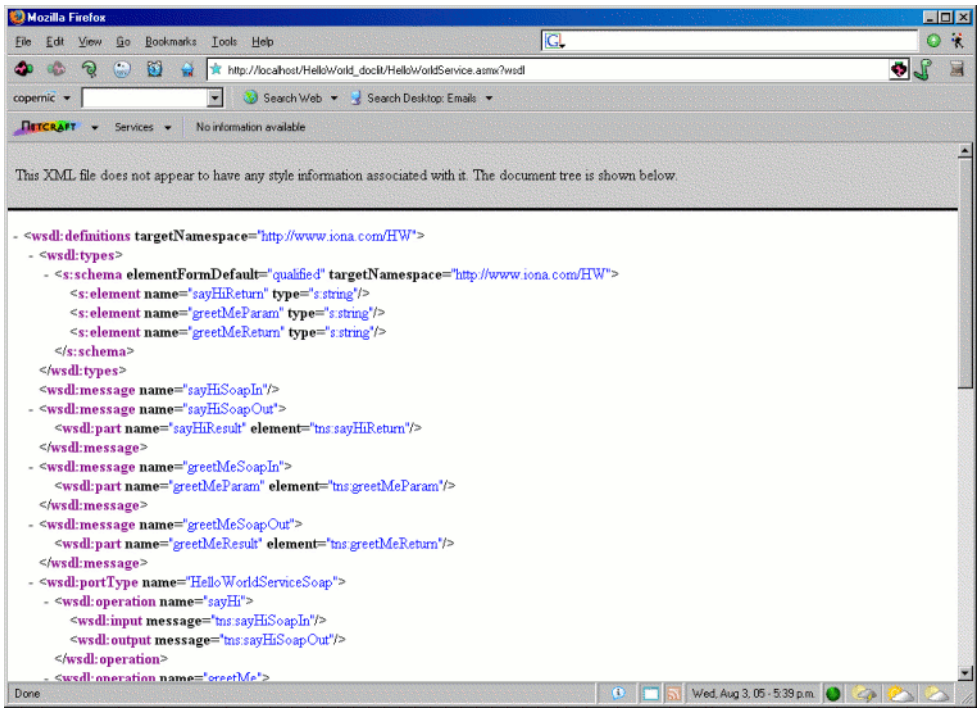


**Figure 2:** *.NET Service WSDL in a Browser*

2. Save the WSDL document to your local system using your browser's **File|Save As** option.
3. If you are using Artix Designer, import the .NET service's WSDL into an Artix project.
4. Generate client code stubs as you would for any Artix client.
5. Develop your client using standard Artix APIs.

# Using Artix to Bridge from a Backend Service to .NET

**Overview**

In situations where you want to build .NET clients that can access services that are provided by backend servers that do not have a native SOAP/HTTP interface, you can use Artix as a bridge, as illustrated in Figure 3. This use case is very similar to the use case described in "Service Enabling Backend Services" on page 43. The major difference is that you will use .NET to develop the client, as described in "Building a .NET Client for an Artix Service" on page 66.

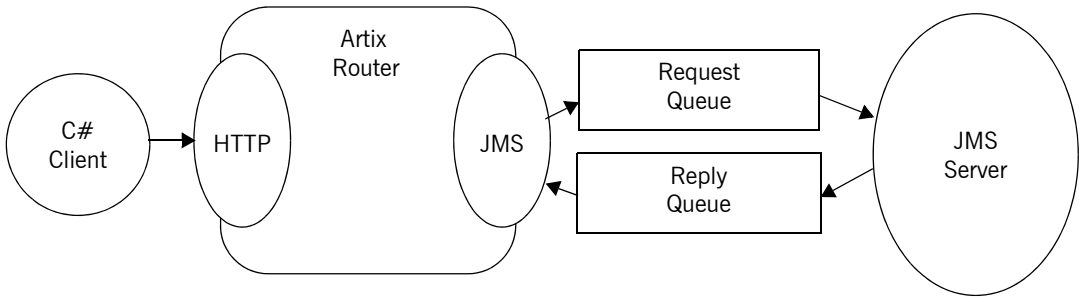


**Figure 3:** *Artix Bridge Between .NET and a JMS Service*

**Procedure**

To build a bridge between a backend service and .NET clients using Artix, do the following:

1. Create an Artix contract describing your backend service as described in "Describing a Service in WSDL" on page 44.
2. Add a SOAP/HTTP endpoint definition to the service contract as described in "Defining a SOAP/HTTP Endpoint" on page 55.

3. Deploy an Artix switch as described in "Configuring and Deploying an Artix Router" on page 58.

   **Note:** If you want to be able to access the contract from the running switch, add `wsdl_publish` to the switch's `orb_plugins` list.

4. Create a new empty C# project in Visual Studio.
5. Add a new Web Reference to your project to bring up the Web Reference browser.
6. In the **Address** field of the browser, type in the location of the service's contract.
   - If the switch is deployed and has the Artix WSDL publishing plug-in configured, you can enter the address of the running switch to get its contract.
   - If you have a copy of the switch's contract stored on an accessible file system, you can enter the full path name of the contract.
7. Once the contract is loaded, click the **Add Reference** button to add the service to your project.
8. Add a new C# class to your project.
9. When developing your client, instantiate a proxy using the name of the SOAP/HTTP endpoint definition in the imported contract.

When Visual Studio builds your client, it automatically generates the proxy code needed to invoke on the backend service using the Artix switch as an intermediary.

# Using Artix Services from .NET clients

**Overview**

Because both the Artix locator and the Artix session manager communicate using SOAP/HTTP, they can be accessed by .NET clients. You could even download their contracts to generate proxies for them. However, the data returned by the Artix services are complex types that are not natively understood by .NET. To overcome this problem, Artix provides a helper library that contains the following:

- A proxy class for the locator.
- A class for representing Artix References.
- A helper class for extracting the SOAP address from an Artix Reference.
- A proxy class for the session manager.

**In this section**

This section discusses the following topics:

# Using the Artix Locator

**Overview**

The Artix locator is a lightweight service for discovering the contact information of a deployed Artix service. While only services developed with Artix can register with the Artix locator, .NET clients can query the Artix locator for services using one of two methods:

- Use a copy of the locator service contract supplied with Artix and the XML Schema definition of an Artix reference, you can build a locator service proxy in .NET and write the logic to extract the appropriate information from the returned Artix reference.
- Use the `Bus.Services.dll` library provided with Artix to access the locator and decipher the returned Artix reference.

It is recommended that you use the latter method for ease of development and to protect your applications from changes in the Artix reference XML Schema definition.

For more information on the Artix locator see Configuring and Deploying Artix Solutions.

**What you need before starting**

Before starting to develop a client that uses the Artix locator to look up live instances of an Artix service, you need the following things:

- The HTTP address of the locator you are going to use.
- A locally accessible copy of a contract defining the service you desire to ultimately invoke upon.

**Procedure**

To use the Artix locator to discover the location of a deployed Artix service from a .NET client, do the following:

1. Create a new project in Visual Studio.
2. Right-click the folder for your new project and select **Add Reference** from the pop-up menu.

3. You will see a window similar to that shown in Figure 4.

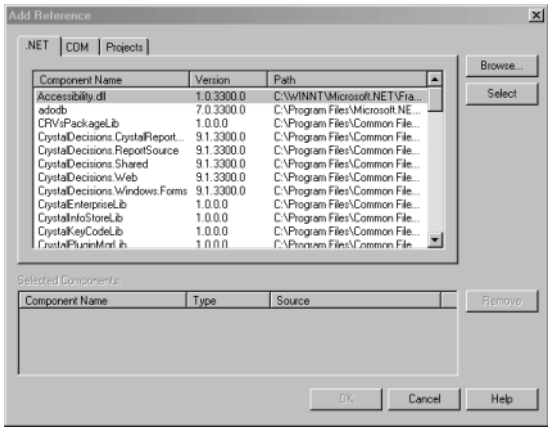


**Figure 4:** *Add Reference Window*

4. Click **Browse.**
5. In the file selection window, browse to your Artix installation and select `utils\.NET\Bus.Services.dll`.
6. Click **OK** to return to the Visual Studio editing area.
7. Right-click the folder for you new project and select **Add Web Reference** from the pop-up menu.
8. In the **Address** field of the browser, enter the full pathname of the contract for the service on which you are going to make requests.
9. Add a new C# class to your project.
10. Add the statement `using Bus.Services;` after the statement `using System;`.
11. Create a service proxy for the Artix locator by instantiating an instance of the `Bus.Services.Locator` class as shown in Example 9.

**Example 9:** *Instantiating a Locator Proxy in .NET*

```
Locator l = new Locator("http://localhost:8080");
```

The string parameter of the constructor is the HTTP address of a deployed Artix locator.

12. Create a QName representing the name of the service you wish to locate using an instance of the `System.Xml.XmlQualifiedName` class as shown in Example 10.

**Example 10:** *Creating a .NET QName*

```
XmlQualifiedName service = new XmlQualifiedName(
  "HelloWorldService",
  "http://www.iona.com/hello_world_soap_http"
);
```

13. Invoke the `lookup_endpoint()` method on the locator proxy as shown in Example 11.

**Example 11:** *Looking Up and Endpoint Reference.*

```
Reference ref = l.lookup_endpoint(service);
```

`lookup_endpoint()` takes the QName of the desired service as a parameter and returns an Artix reference if an instance of the specified service is registered with the locator instance. Artix references are implemented in the .NET `Bus.Services.Reference` class.

14. Create a .NET proxy for the service on which you are going to make requests as you would normally.
15. Change the value of the proxy's `.Url` member to the SOAP address contained in the Artix reference returned from the locator as shown in Example 12.

**Example 12:** *Changing the URL of a .NET Service Proxy to Use a Reference*

```
pxy.Url = ReferenceHelper.GetSoapAddress(ref);
```

The `Bus.Services.ReferenceHelper.GetSoapAddress()` method extracts the SOAP address from an Artix reference and returns it as a string.

16. Make requests on the service as you would normally.

## Example

Example 13 shows the code for a .NET client that looks up the HelloWorld service from a locator deployed at `localhost:8080`.

**Example 13:** *.NET Client Using the Artix Locator*

```
using System;
using Bus.Services;

namespace HelloWorldClient
{
  class Client
  {
    static void Main(string[] args)
    {
     Locator l = new Locator("http://localhost:8080");

     XmlQualifiedName service = new XmlQualifiedName(
                    "HelloWorldService",
                   "http://www.iona.com/hello_world_soap_http");

      Reference ref = l.lookup_endpoint(service);

      HelloWorldService proxy = new HelloWorldService();
      proxy.Url = ReferenceHelper.GetSoapAddress(ref);

      string str_out, str_in;

      str_out = proxy.sayHi();
      Console.WriteLine("sayHi method returned: " + str_out);

      str_in = "Early Adopter";
      str_out = proxy.greetMe(str_in);
      Console.WriteLine("greetMe method returned: " + str_out);
    }
  }
}
```

# Using the Artix Session Manager

**Overview**

The Artix session manager is a lightweight service that manages the number of concurrent clients that access a group of services. As with the Artix locator, only Artix services can register with a session manager. However, .NET clients can use the session manager to make requests on managed services using the `Bus.Services.dll` library.

Working with the Artix session manager is slightly trickier than working with the Artix locator. This is because the Artix session manager uses SOAP headers to pass session tokens between clients and services. The session manager also has a number of methods for managing active sessions.

The helper classes included in the `Bus.Services` library simplify working with the session manager by providing native .NET calls to access the session manager. They also handle session renewal and attaching session headers to outgoing requests.

For more information on the Artix session manager, see Configuring and Deploying Artix Solutions.

**What you need before starting**

Before starting to develop a client that uses the Artix session manager you need:

- A means for contacting a deployed Artix session manager. This can be one of the following:
  - An Artix reference
  - An HTTP address
  - A local copy of the session manager WSDL contract
- A locally accessible copy of the WSDL contract that defines the service that you want the client to invoke upon.
- To install WSE 2.0 SP3 before starting an Artix .NET session manager client.

**Procedure**

To develop a .NET client that uses the Artix session manager, do the following:

1. Create a new project in Visual Studio.
2. Right-click the folder for you new project and select **Add Reference** from the pop-up menu.
3. Click **Browse** on **Add Reference** window.
4. In the file selection window browse to your Artix installation and select the `Bus.Services.dll` from the *`InstallDir`*`\artix\`*`Version`*`\utils\.NET` directory.
5. Click **OK** to return to the Visual Studio editing area.
6. Right-click the folder for your new project and select **Add Web Reference** from the pop-up menu.
7. In the **Address:** field of the browser, enter the full pathname of the contract for the service on which you are going to make requests.
8. Click **Add Reference** to return to the Visual Studio editing area.
9. Open the `.cs` file generated for the contract you imported.
10. Locate the class declaration for the service on which you intend to make requests. The class declaration will look similar to that shown in Example 14.

**Example 14:** *.Net Service Proxy Class Declaration*

```
public class SOAPService :
   System.Web.Services.Protocols.SoapHttpClientProtocol {
```

11. Change the class' base type from `System.Web.Services.Protocols.SoapHttpClientProtocol` to `Microsoft.Web.Services2.WebServicesClientProtocol`. The resulting class declaration will look similar to that shown in Example 15.

**Example 15:** *.Net Session Managed Proxy Class Declaration*

```
public class SOAPService :
   Microsoft.Web.Services2.WebServicesClientProtocol {
```

Reassigning the service proxy class to the Artix specific base class adds methods to the proxy that allow it to work with the session manager.

12. Add a new C# class to your project.
13. Add the statement `using Bus.Services;` after the statement `using System;`.
14. Create a service proxy for the Artix session manager by instantiating an instance of the `Bus.Services.SessionManager` class as shown in Example 16.

**Example 16:** *Instantiating a Session Manager Proxy in .Net*

```
SessionManager sessionManager = new SessionManager
    ("http://localhost:9007/services/sessionManagement/
     sessionManagerService");
```

The constructor's parameter is the HTTP address of a deployed session manager. The `SessionManager` class also has a construct that takes an Artix reference for use with the Artix locator.

15. Create a new Artix session by instantiating an instance of `Bus.Services.Session` as shown in Example 17.

**Example 17:** *Creating a New Session*

```
Session session = new Session(sessionManager, "SM_Demo", 20);
```

The constructor takes three parameters:

- An instantiated `SessionManager` object.
- A string identifying the group for which the client wants a session; in this example, the group name is `SM_Demo`.
- The default timeout value, in seconds, for the session.

Once the session is created, the session will automatically attempt to renew itself until the session is closed. The client does not need to worry about renewing the session.

**Note:** If you want your client to manually renew the session, you can use `sm.DoAutomaticSessionRenew(false)` to turn off automatic session renewal. To renew a session programatically call `renewSession()` on the `Session` object.

16. Get a list of the references for the endpoints that are in the session's group using the `SessionManager.get_all_endpoints()` function as shown in Example 18.

**Example 18:** *Getting the Endpoint References*

```
Bus.Services.Types.EndpointReferenceType[] refs =
    sessionManager.getAllServiceEndpoints(sessionId);
```

The `get_all_endpoints()` function takes the session ID of the session and returns an array of Artix references. Each entry in the array contains the endpoint of one member of the group for which the session was requested.

17. Create a .Net proxy for the service on which you are going to make requests as you normally would.
18. Change the value of the proxy's `.Url` member to the SOAP address of one of the Artix references returned from the session manager as shown in Example 19.

**Example 19:** *Changing the URL of a .Net Service Proxy to Use a Reference*

```
simpleService.Url = refs[0].Address.Value;
```

How you determine which member of the returned array contains the desired endpoint is an implementation detail beyond the scope of this discussion.

19. Instruct the proxy to include the session header in all of its requests by adding a session filter on the proxy output SOAP filters as shown in Example 20.

**Example 20:** *Setting a Proxy's Session Header*

```
simpleService.Pipeline.OutputFilters.Add(new
    Bus.Services.SessionFilter(session));
```

Once you have made the above call, all requests made by the proxy will contain an Artix session header. The session manager uses the session header to validate the client's requests against the list of valid sessions.

20. Make requests on the service as you would normally.
21. When you are done with the service, end the session by calling `EndSession()` on the `session` object, as shown in Example 21:

**Example 21:** *Ending a Session*

```
session.EndSession()
```

## Example

Example 22 shows the code for a .NET client that makes requests on a HelloWorld service that is managed by a session manager deployed at `localhost:8080`.

**Example 22:** *.NET Client Using the Artix Session Manager*

```
using System;
using Bus.Services;

namespace HelloWorldClient
{
  class Client
  {
    static void Main(string[] args)
    {
      SessionManager sm = new
   SessionManager("http://localhost:8080");

      Session s = new Session(sm, "theGroup", 60);

      SOAPService proxy = new SOAPService();
      Reference[] r = sm.get_all_endpoints(s.GetSessionId());
      proxy.Url = r[0].Address.Value;;

      proxy.Pipeline.OutputFilters.Add(new
   Bus.Services.SessionFilter(session));

      string str_out, str_in;

      str_out = proxy.sayHi();
      Console.WriteLine("sayHi method returned: " + str_out);

      str_in = "Early Adopter";
      str_out = proxy.greetMe(str_in);
      Console.WriteLine("greetMe method returned: " + str_out);

      s.EndSession();
    }
  }
}
```

CHAPTER 6

# Using Artix with WebSphere MQ

*To make Artix an WebSphere MQ interoperate you must properly set up your environment.*

## Overview

When working with Artix and WebSphere MQ, there are five deployment scenarios:

1. Artix client and server are on the same host computer as a full WebSphere MQ installation.
2. Artix client and server are on different host computers and each host has a full WebSphere MQ installation.
3. The Artix client is installed on a host computer with a client WebSphere MQ. The Artix server is installed on a computer with a full WebSphere MQ installation.
4. Artix client and server are installed on a host computer with a client WebSphere MQ installation. A full WebSphere MQ installation is on a remote host computer. The remote WebSphere MQ installation manages the messages to and from the client and server processes.
5. Artix client and server processes are installed on a host computer with have a full WebSphere MQ installation. A full WebSphere MQ installation is on remote host computer. The remote WebSphere MQ installation manages the messages to and from the client or server process.

The following discussions are based on using WebSphere MQ and Artix installed on Windows computers. In order to successfully deploy any of the scenarios involving multiple computers, log into each of the computers with the same username and password.

**In this chapter**

This chapter discusses the following topics:

# Artix and MQ on a Single Computer

**Overview**

In this scenario, all of the parts of your Artix solution are on a single computer. The Artix client and server are deployed onto a computer that has a full Websphere MQ installation. Both the client and server use a common queue manager and pair of local queues.

**WebSphere MQ Setup**

To set up the WebSphere MQ environment do the following:

1. Create a queue manager.
2. Create a local queue of normal usage to be used as the request queue.
3. Create a local queue of normal usage to be used as the response queue.

**Contract**

In the contract fragment shown in Example 23, the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

**Example 23:** *Contract for a Simple Websphere MQ Integration*

```
<service name="MQService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <mq:client  QueueManager="MY_DEF_QM"
                QueueName="HW_REQUEST"
                ReplyQueueManager="MY_DEF_QM"
                ReplyQueueName="HW_REPLY"
                AccessMode="send"
                CorrelationStyle="correlationId" />
    <mq:server  QueueManager="MY_DEF_QM"
                QueueName="HW_REQUEST"
                ReplyQueueManager="MY_DEF_QM"
                ReplyQueueName="HW_REPLY"
                AccessMode="receive"
                CorrelationStyle="correlationId" />
  </port>
</service>
```

# Client and Server on Different MQ Servers

**Overview**

This scenario requires a more extensive configuration of the WebSphere MQ installations. In each installation, a queue manager contains both local and remote queues. The client and server interact with the local queues while the remote queues are responsible for passing messages between the two WebSphere MQ installations. The contract contains entries for both local and remote queues.

**Client WebSphere MQ setup**

To set up the Websphere MQ environment on the Artix client's host do the following:

1. Create a queue manager.

   **Note:** The name of the queue manager should be a different name on each host.

2. Create a local queue of usage `transmission`.
3. Create a local queue of usage `normal` to be the response queue.
4. Create a remote queue to reference the local queue on which the server will receive requests.

```
Queue Name                      OutgoingRequest
Remote Queue Name               IncomingRequest
Remote Queue Manager Name       server queue manager
Transmission Queue Name         local transmission queue
```

5. Create a remote queue to reference the remote queue on which the server will post replies.

```
Queue Name                      ServerReplyQueue
Remote Queue Name               OutgoingReply
Remote Queue Manager Name       server queue manager
Transmission Queue Name         local transmission queue
```

6. Create a receiver channel to receive replies from the server.

```
Channel Name                    CHANNEL_S_C
```

| | |
|---|---|
| Transmission Protocol | TCP/IP |

7. Create a sender channel to send requests to the server.

| | |
|---|---|
| Channel Name | CHANNEL_C_S |
| Transmission Protocol | TCP/IP |
| Connection Name | hostname of server |
| Transmission Queue | local transmission queue |

---

**Server WebSphere MQ setup**

To set up the WebSphere MQ environment on the Artix server's host computer do the following:

1. Create a queue manager.
2. Create a local queue of usage `transmission`.
3. Create a local queue of usage `normal` to receive requests from the client.
4. Create a remote queue to refer to the local queue on which the client will receive replies.

| | |
|---|---|
| Queue Name | OutgoingReply |
| Remote Queue Name | IncomingReply |
| Remote Queue Manager Name | client's queue manager |
| Transmission Queue Name | local transmission queue |

5. Create a receiver channel to receive requests from the client.

| | |
|---|---|
| Channel Name | CHANNEL_C_S |
| Transmission Protocol | TCP/IP |

6. Create a sender channel to send replies to the client.

| | |
|---|---|
| Channel Name | CHANNEL_S_C |
| Transmission Protocol | TCP/IP |
| Connection Name | hostname of client computer |
| Transmission Queue | local transmission queue |

### Contract

In the contract fragment shown in Example 24, the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

**Example 24:** *Contract for Remote WebSphere MQ Set Up*

```
<wsdl:service name="MQService">
  <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <mq:client AccessMode="send"
               QueueManager="QMgrC"
               QueueName="OutgoingRequest"
               ReplyQueueManager="QMgrC"
               ReplyQueueName="IncomingReply"
               CorrelationStyle="correlationId"
               AliasQueueName="ServerReplyQueue" />
    <mq:server AccessMode="receive"
               QueueManager="QMgrS"
               QueueName="IncomingRequest"
               ReplyQueueManager="QMgrS"
               ReplyQueueName="OutgoingReply"
               CorrelationStyle="correlationId" />
  </wsdl:port>
</wsdl:service>
```

The client transport is configured to use the WebSphere MQ installation on the computer that hosts the client application. The server transport is configured to use the WebSphere MQ installation on the computer that hosts the server application. The additional transport attribute, `AliasQueueName`, is required in the `<mq:client>` element. This attribute insures that the server process uses its remote queue, `OutgoingReply`, and queue manager, `QMgrS`, when posting the response. Without this entry, the server would attempt to post the response to the reply queue identified in the MQ message header. The MQ message header content is derived from the information in the `<mq:client>` element.

# Client on MQ Client and Server on MQ Server

**Overview**

This scenario requires a queue manager and two local queues. In addition, a Server Transport Channel must be defined for the queue manager.

On the client's host there is a client WebSphere MQ installation. There are no queue managers or queues configured on this host. Before running the Artix client application, the `MQSERVER` environment variable must be set. This variable identifies the Server Transport Channel, the computer hosting the full installation of WebSphere MQ, and the TCP/IP port used by the queue manager listener.

On the server's host, there is a full WebSphere MQ installation.

**WebSphere MQ client setup**

Install the client WebSphere MQ application on the computer that will run the Artix client application. The installation process places the WebSphere MQ libraries onto the system path. Prior to running the client application, the environment variable `MQSERVER` must be set in the command window.

**WebSphere MQ server setup**

To set up the WebSphere MQ environment on the server's host computer do the following:

1. Create a queue manager.
2. Create a local queue of usage `normal` onto which the requests will be placed.
3. Create a local queue of usage `normal` onto which the responses will be placed.
4. Create a server connection channel.

```
Channel Name                    CONNECT1
Protocol Type                   TCP/IP
```

**Operation**

When you are ready to start-up your application do the following:

1. Start the queue manager on the WebSphere MA server.
2. Start the Artix server.

3. Start the Artix client.
   i. Open a command window.
   ii. Set the environment variable:

```
set MQSERVER=CONNECT1/TCP/server_hostname
```

   iii. Run the client executable.

---

**Contract**

In the contract fragment shown in Example 25 the `<mq:client>` and the `<mq:server>` elements include the attributes that configure the transport.

**Example 25:** *Port Settings for a Client and Server*

```
 <service name="MQService">
   <port binding="tns:Greeter_SOAPBinding" name="MQPort">
     <mq:client QueueManager="MY_DEF_QM"
                QueueName="HW_REQUEST"
                ReplyQueueManager="MY_DEF_QM"
                ReplyQueueName="HW_REPLY"
                AccessMode="send"
                CorrelationStyle="correlationId" />
     <mq:server QueueManager="MY_DEF_QM"
                QueueName="HW_REQUEST"
                ReplyQueueManager="MY_DEF_QM"
                ReplyQueueName="HW_REPLY"
                AccessMode="receive"
                CorrelationStyle="correlationId" />
   </port>
</service>
```

# Using a Remote MQ Server

**Overview**

This scenario is similar to scenario described in "Client on MQ Client and Server on MQ Server" on page 91 with the exception that the computer hosting the server process is not the same computer on which the WebSphere MQ is installed. The Artix client and the Artix server are run on remote computers. They can be on either the same computer or on different computers.

The computer(s) used to run the Artix processes must have a client WebSphere MQ installation. There are no queue managers or queues configured on this computer. Before running the Artix processes, the `MQSERVER` environment variable must be set. This variable identifies the Server Transport Channel, the computer hosting the full installation of WebSphere MQ, and the TCP/IP port used by the queue manager listener.

**Remote system set up**

Install the client WebSphere MQ application on the computer that will run the Artix client and/or the Artix server applications. The installation process places the WebSphere MQ libraries onto the system path. Prior to running the Artix applications, the environment variable `MQSERVER` must be set in the command window.

**WebSphere MQ server set up**

To set up the system on which the WebSphere MQ queue manager will run do the following:

1. Create a queue manager.
2. Create a local queue of usage `normal` onto which the requests will be placed.
3. Create a local queue of usage `normal` onto which the responses will be placed.
4. Create a server connection channel.

```
Channel Name                    CONNECT1
Protocol Type                   TCP/IP
```

**Operation**

When you are ready to start-up your application do the following:

1. Start the queue manager.
2. Start the server.
   i. Open a command window.
   ii. Set the environment variable:

```
set MQSERVER=CONNECT1/TCP/MQ_hostname
```

   iii. Run the server process.
3. Start the client.
   i. Open a command window.
   ii. Set the environment variable:

```
set MQSERVER=CONNECT1/TCP/MQ_hostname
```

   iii. Run the client process.

**Contract**

In the contract fragment shown in Example 26 the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

**Example 26:** *Contract for a Remote WebSphere MQ Server*

```
<service name="MQService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <mq:client QueueManager="MY_DEF_QM"
               QueueName="HW_REQUEST"
               ReplyQueueManager="MY_DEF_QM"
               ReplyQueueName="HW_REPLY"
               AccessMode="send"
               CorrelationStyle="correlationId" />
     <mq:server QueueManager="MY_DEF_QM"
                QueueName="HW_REQUEST"
                ReplyQueueManager="MY_DEF_QM"
                ReplyQueueName="HW_REPLY"
                AccessMode="receive"
                CorrelationStyle="correlationId" />
  </port>
</service>
```

# Using a Remote MQ Server from Full MQ Installations

**Overview**

This scenario is similar to the one described in "Using a Remote MQ Server" on page 93 with the exception that the computer(s) hosting the client or server process also has a full WebSphere MQ installation. However, there are no queue managers or queues configured on this host.

**Additional set up**

The only difference in the set up is that you need to specify an additional MQ port attribute for any of the Artix applications running on a system with a full WebSphere MQ installation. The `Server_Client="client"` attribute setting informs the Artix runtime that it needs to load the Websphere MQ client libraries instead of the Websphere MQ server libraries.

**Contract**

In the contract fragment shown in Example 27 the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

**Example 27:** *using Artix on a Full MQ Installation*

```
<service name="MQService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <mq:client QueueManager="MY_DEF_QM"
               QueueName="HW_REQUEST"
               ReplyQueueManager="MY_DEF_QM"
               ReplyQueueName="HW_REPLY"
               AccessMode="send"
               CorrelationStyle="correlationId"
               Server_Client="client" />
    <mq:server QueueManager="MY_DEF_QM"
               QueueName="HW_REQUEST"
               ReplyQueueManager="MY_DEF_QM"
               ReplyQueueName="HW_REPLY"
               AccessMode="receive"
               CorrelationStyle="correlationId"
               Server_Client="client" />
  </port>
</service>
```

CHAPTER 7

# Writing XSLT Scripts for the Artix Transformer

*The Artix transformer is a light weight service that uses XSLT scripts to transform messages sent from Artix endpoints.*

**Overview**

The transformer uses a WSDL file, which describes the source message and the target message, to create in-memory XML representations of the input and output data. Then, using an XSLT script that describes the mapping of the input message to output message the transformer creates the output message.

Since you do not have a printout of the in-memory XML representations, writing the XSLT script file appears to be a difficult task. However, you have two sources of information that will guide you: the WSDL file and the content of the SOAP message body.

**Note:** The SOAP message body is only used as a crutch in developing your XSLT scripts. The Artix transformer can work with any of the bindings and transports supported by Artix.

**In this chapter**

This chapter discusses the following topics:

# The XSLT Script Template

**Overview**

It is probably not apparent from the transformation demo that ships with Artix that there is a common format to all XSLT scripts. This is due to the fact that the incoming and outgoing messages include a single part. The in-memory XML representations of these messages, all of the content that you need to manipulate will be included as child elements under the element that corresponds to the message part. Therefore, it is fairly straightforward to set up the basic content of the XSLT script.

**Sections of the script**

There are three areas of interest in an XSLT script:

1. This area identifies the node from which to begin the transformation. When using the transformer, the root node `-/-` is used as the starting point to indicate that processing should begin from the beginning of the in-memory representation.
2. This area describes the syntax of the transformed message. It is always contained in a `message` element. The trick in writing this part of the script is determining what elements to place as children to the `message` element. The name of the children of the `message` element is determined by the `name` attribute of the `part` elements of the WSDL `message` element specified as the input message of the invoked operation. If you examine the WSDL documents in this chapter, you will note that the `transformer_reply` element corresponds to the value of the `name` attribute for the part within the operation's output message. That is, for the interface `transformer`, the operation `transform` has an output message of type `transformer_reply_message`. The part within this message is named `transformer_reply`, and this becomes the value for the name of the element defining the reply message's content. The element contains a single `apply-templates` element.
3. This section contains the processing directives that describes the output for a template match. The primary question is what value to assign to the `match` attribute of the `template` element. This value is derived from the `name` attribute of the part within the operation's input

message. That is, for the interface `transformer`, the operation `transform` has an input message of type `client_request_message`. The part within this message is named `client_request`, and this becomes the value for the match attribute of the opening template tag that delimits the transformation commands.

### The template

Example 28 shows the template for the XSLT scripts used later in the chapter.

**Example 28:** *XSLT Script Template*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
1   <xsl:template match="/">
2   <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
    </xsl:template>
3   <xsl:template match="client_request">
      . . .
    </xsl:template>
</xs1:transform>
```

# Transforming a Sequence into a String

**Overview**

In this example, the input data is in a sequence that includes two members and the output data is a simple string.

**The contract**

The logical section of the WSDL file is shown in Example 29.

**Example 29:** *Contract Fragment for Sequence to String*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="client"
             targetNamespace="http://www.iona.com/xslt"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:http-conf=
          "http://schemas.iona.com/transports/http/configuration"
             xmlns:ns1="http://www.iona.com/xslt/corba/typemap/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://www.iona.com/xslt"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <schema targetNamespace="http://www.iona.com/xslt"
           xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <element name="transform"
               type="tns:complex_request_type"/>
      <element name="transformResponse"
               type="xsd:string"/>
      <complexType name="complex_request_type">
        <sequence>
          <element name="first_name" type="xsd:string"/>
          <element name="last_name" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  <message name="client_request_message">
    <part element="tns:transform" name="client_request"/>
  </message>
```

**Example 29:** *Contract Fragment for Sequence to String*

```
  <message name="transformer_reply_message">
    <part element="tns:transformResponse"
          name="transformer_reply"/>
  </message>
  <portType name="transformer">
    <operation name="transform">
      <input message="tns:client_request_message"
             name="transformRequest"/>
      <output message="tns:transformer_reply_message"
              name="transformResponse"/>
    </operation>
  </portType>
```

**The XSLT script**

Example 30 shows an XSLT script that simply concatenates the first and last name values from the incoming data into the output data string.

**Example 30:** *Simple XSLT Script*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
  <xsl:template match="client_request">
    <xsl:value-of select="first_name"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="last_name"/>
  </xsl:template>
</xsl:transform>
```

Note the correspondence between the bold type font in the WSDL document and XSLT script. If you were to examine the content of the SOAP body, you would see that the strings corresponding to the first and last names are simply contained in the `first_name` and `last_name` elements, which is why `first_name` and `last_name` were specified as the values of the `select` attributes.

# Modifying a Simple Sequence

**Overview**

In this example, both the input and output data are sequences. However, the output sequence contains one fewer member. The transformation involves mapping one member of the input data into a corresponding member in the output data, and mapping the concatenation of the other two input data members into the second member in the output sequence.

**The contract**

The logical section of the WSDL file is shown in Example 31.

**Example 31:** *Contract Fragment for Modifying a Sequence*

```
<types>
  <schema targetNamespace="http://www.iona.com/xslt"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="transform"
             type="tns:complex_request_type"/>
    <element name="transformResponse"
             type="tns:complex_response_type"/>
    <complexType name="complex_request_type">
      <sequence>
        <element name="first_name" type="xsd:string"/>
        <element name="last_name" type="xsd:string"/>
        <element name="postal_code" type="xsd:string"/>
      </sequence>
    </complexType>
    <complexType name="complex_response_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="zipcode" type="xsd:string"/>
        <element maxOccurs="1" minOccurs="1"
                 name="full_name" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
</types>
  <message name="client_request_message">
    <part element="tns:transform" name="client_request"/>
  </message>
```

**Example 31:** *Contract Fragment for Modifying a Sequence*

```
  <message name="transformer_reply_message">
    <part element="tns:transformResponse"
          name="transformer_reply"/>
  </message>
  <portType name="transformer">
    <operation name="transform">
      <input message="tns:client_request_message"
             name="transformRequest"/>
      <output message="tns:transformer_reply_message"
              name="transformResponse"/>
    </operation>
  </portType>
```

**The script**

The XSLT script shown in Example 32 copies the value of the `postal_code` element of the input message into the `zipcode` element the output message. It then performs the same concatenation as in the first example.

**Example 32:** *Sequence to Sequence XSLT Script*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
  <xsl:template match="client_request">
    <zipcode>
      <xsl:value-of select="postal_code"/>
    </zipcode>
    <full_name>
      <xsl:value-of select="first_name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="last_name"/>
    </full_name>
  </xsl:template>
</xsl:transform>
```

Unlike the previous example where the output message was a single element, the output message in this example contains multiple elements. To specify the extra complexity, the script specifies processing directives for each of the elements contained in the output message.

These processing instructions may appear confusing because there are elements derived from the content of the output message and `select` attributes that reference content from the input message. This is because the transformer processes the directives in the following way:

1. Create the opening tag for a `zipcode` element in the output message.
2. Find the 1st instance of a `postal_code` element in the input message and place its value inside the `zipcode` element.
3. Create the closing tag for a `zipcode` element in the output message.
4. Create the opening tag for a `full_name` element in the output message.
5. Find the 1st instance of a `first_name` element in the input message and place its value inside the `ful_name` element.
6. Place a space after the first name.
7. Find the 1st instance of a `last_name` element in the input message and place its value after the space.
8. Create the closing tag for a `full_name` element in the output message.

# Working with Nested Input Sequences

**Overview**

In this example, the input data is a sequence that includes two child sequences. The output data is a sequence with two members.

**The contract**

The logical section of the WSDL file is shown in Example 33.

**Example 33:** *Contract Fragment for Nested Input Sequences*

```
<types>
  <schema targetNamespace="http://www.iona.com/xslt"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="transform" type="tns:complex_request_type"/>
    <element name="transformResponse"
             type="tns:complex_response_type"/>
    <complexType name="name_type">
      <sequence>
        <element name="first_name" type="xsd:string"/>
        <element name="last_name" type="xsd:string"/>
      </sequence>
    </complexType>
    <complexType name="specifics_type">
      <sequence>
        <element name="age" type="xsd:string"/>
        <element name="postal_code" type="xsd:string"/>
      </sequence>
    </complexType>
    <complexType name="complex_request_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="individual"
                 type="tns:name_type"/>
        <element maxOccurs="1" minOccurs="1"
                 name="personal_details"
                 type="tns:specifics_type"/>
      </sequence>
    </complexType>
```

**Example 33:** *Contract Fragment for Nested Input Sequences*

```
    <complexType name="complex_response_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="zipcode" type="xsd:string"/>
        <element maxOccurs="1" minOccurs="1"
                 name="full_name" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
</types>
  <message name="client_request_message">
    <part element="tns:transform" name="client_request"/>
  </message>
  <message name="transformer_reply_message">
    <part element="tns:transformResponse"
          name="transformer_reply"/>
  </message>
  <portType name="transformer">
    <operation name="transform">
      <input message="tns:client_request_message"
             name="transformRequest"/>
      <output message="tns:transformer_reply_message"
              name="transformResponse"/>
    </operation>
  </portType>
```

**The XSLT script**

The XSLT script for this example, shown in Example 34, is very similar to the script in Example 32. The major difference is that you access the input data by referencing the nested sequences. For example, to match the `postal_code` element you use the expression `personal_deatils/postal_code`.

**Example 34:** *XSLT Script for a Nested Input Sequence*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:ns1="http://www.iona.com/xslt"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
```

**Example 34:** *XSLT Script for a Nested Input Sequence*

```
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
  <xsl:template match="client_request">
    <zipcode>
      <xsl:value-of select="personal_details/postal_code"/>
    </zipcode>
    <full_name>
      <xsl:value-of select="individual/first_name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="individual/last_name"/>
    </full_name>
  </xsl:template>
</xsl:transform>
```

# Working with Attributes in an Input Message

**Overview**

In this example, the input message is a sequence with two child sequences. One of the child sequences includes an attribute. In this transformation, the important concept is how the attribute is accessed in the processing instructions. The value of the attribute is placed into one of the members of the output sequence.

**The contract**

The logical section of the WSDL file is shown in Example 35.

**Example 35:** *Contract Fragment for Input Sequences with Attributes*

```
<types>
  <schema targetNamespace="http://www.iona.com/xslt"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="transform" type="tns:complex_request_type"/>
    <element name="transformResponse"
             type="tns:complex_response_type"/>
    <complexType name="name_type">
      <sequence>
        <element name="first_name" type="xsd:string"/>
        <element name="last_name" type="xsd:string"/>
      </sequence>
    </complexType>
    <complexType name="specifics_type">
      <sequence>
        <element name="age" type="xsd:string"/>
        <element name="postal_code" type="xsd:string"/>
      </sequence>
      <attribute name="sex" type="xsd:string"/>
    </complexType>
    <complexType name="complex_request_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="individual"
                 type="tns:name_type"/>
        <element maxOccurs="1" minOccurs="1"
                 name="personal_details"
                 type="tns:specifics_type"/>
      </sequence>
    </complexType>
```

**Example 35:** *Contract Fragment for Input Sequences with Attributes*

```
    <complexType name="complex_response_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="sex" type="xsd:string"/>
        <element maxOccurs="1" minOccurs="1"
                 name="full_name" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
</types>
<message name="client_request_message">
  <part element="tns:transform" name="client_request"/>
</message>
<message name="transformer_reply_message">
  <part element="tns:transformResponse"
        name="transformer_reply"/>
</message>
<portType name="transformer">
  <operation name="transform">
    <input message="tns:client_request_message"
           name="transformRequest"/>
    <output message="tns:transformer_reply_message"
            name="transformResponse"/>
  </operation>
</portType>
```

### The XSLT Script

While the syntax for accessing the members of the nested sequence in the XSLT script shown Example 36 looks similar to the syntax used in Example 34, notice the `@` used when referencing the `sex` attribute. The `@` specifies that the value to be matched in an attribute, not an element.

**Example 36:** *XSLT Script for Accessing Attributes*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:ns1="http://www.iona.com/xslt"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
```

**Example 36:** *XSLT Script for Accessing Attributes*

```
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
  <xsl:template match="client_request">
    <sex>
      <xsl:value-of select="personal_details/@sex"/>
    </sex>
    <full_name>
      <xsl:value-of select="individual/first_name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="individual/last_name"/>
    </full_name>
  </xsl:template>
</xsl:transform>
```

# Working with Attributes in the Output Message

**Overview**

In this example, the output message includes a sequence, with one member, and an attribute, which contains data that was in an attribute within the input message. The important concept in this example is how the processing instructions define the attribute in the output message.

**The contract**

The logical section of the WSDL file is shown in Example 37.

**Example 37:** *Contract Fragment for Output Sequences with Attributes*

```
<types>
  <schema targetNamespace="http://www.iona.com/xslt"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="transform" type="tns:complex_request_type"/>
    <element name="transformResponse"
             type="tns:complex_response_type"/>
    <complexType name="name_type">
      <sequence>
        <element name="first_name" type="xsd:string"/>
        <element name="last_name" type="xsd:string"/>
      </sequence>
    </complexType>
    <complexType name="specifics_type">
      <sequence>
        <element name="age" type="xsd:string"/>
        <element name="postal_code" type="xsd:string"/>
      </sequence>
      <attribute name="sex" type="xsd:string"/>
    </complexType>
```

**Example 37:** *Contract Fragment for Output Sequences with Attributes*

```
    <complexType name="complex_request_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="individual"
                 type="tns:name_type"/>
        <element maxOccurs="1" minOccurs="1"
                 name="personal_details"
                 type="tns:specifics_type"/>
      </sequence>
    </complexType>
    <complexType name="complex_response_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="full_name" type="xsd:string"/>
      </sequence>
      <attribute name="sex" type="xsd:string"/>
    </complexType>
  </schema>
</types>
<message name="client_request_message">
  <part element="tns:transform" name="client_request"/>
</message>
<message name="transformer_reply_message">
  <part element="tns:transformResponse"
        name="transformer_reply"/>
</message>
<portType name="transformer">
  <operation name="transform">
    <input message="tns:client_request_message"
           name="transformRequest"/>
    <output message="tns:transformer_reply_message"
            name="transformResponse"/>
  </operation>
</portType>
```

### The XSLT Script

Example 38 shows an XSLT script that places the `sex` attribute from the input message into the `sex` attribute of the output message. This is done using the `xsl:attribute` directive.

**Example 38:** *Placing an Attribute in an Output Message*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:ns1="http://www.iona.com/xslt"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
  <xsl:template match="client_request">
    <xsl:attribute name="sex">
      <xsl:value-of select="personal_details/@sex"/>
    </xsl:attribute>
    <full_name>
      <xsl:value-of select="individual/first_name"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="individual/last_name"/>
    </full_name>
  </xsl:template>
</xsl:transform>
```

# Working with Nested Sequences in an Output Message

**Overview**

In this example, the output message includes an attribute and a sequence that itself contains a sequence.

**The contract**

The logical section of the WSDL file is shown in Example 39.

**Example 39:** *Contract Fragment for Nested Output Sequences*

```
<types>
  <schema targetNamespace="http://www.iona.com/xslt"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="transform" type="tns:complex_request_type"/>
    <element name="transformResponse"
             type="tns:complex_response_type"/>
    <complexType name="name_type">
      <sequence>
        <element name="first_name" type="xsd:string"/>
        <element name="last_name" type="xsd:string"/>
      </sequence>
    </complexType>
    <complexType name="specifics_type">
      <sequence>
        <element name="age" type="xsd:string"/>
        <element name="postal_code" type="xsd:string"/>
      </sequence>
      <attribute name="sex" type="xsd:string"/>
    </complexType>
    <complexType name="complex_request_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="individual"
                 type="tns:name_type"/>
        <element maxOccurs="1" minOccurs="1"
                 name="personal_details"
                 type="tns:specifics_type"/>
      </sequence>
    </complexType>
```

**Example 39:** *Contract Fragment for Nested Output Sequences*

```
    <complexType name="complex_response_type">
      <sequence>
        <element maxOccurs="1" minOccurs="1"
                 name="full_name" type="tns:name_type"/>
      </sequence>
      <attribute name="sex" type="xsd:string"/>
    </complexType>
  </schema>
</types>
<message name="client_request_message">
  <part element="tns:transform" name="client_request"/>
</message>
<message name="transformer_reply_message">
  <part element="tns:transformResponse"
        name="transformer_reply"/>
</message>
<portType name="transformer">
  <operation name="transformer_operation">
    <input message="tns:client_request_message"
           name="transformRequest"/>
    <output message="tns:transformer_reply_message"
            name="transformResponse"/>
  </operation>
</portType>
```

**The XSLT script**

Unlike in the previous examples, the XSLT script required to create the output message, shown in Example 40, must now include tags to create the `first_name` and `last_name` elements derived from the nested sequence.

**Example 40:** *Creating a Nested Output Sequence*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:ns1="http://www.iona.com/xslt"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
```

**Example 40:** *Creating a Nested Output Sequence*

```
  <xsl:template match="client_request">
    <xsl:attribute name="sex">
      <xsl:value-of select="personal_details/@sex"/>
    </xsl:attribute>
    <full_name>
      <first_name>
        <xsl:value-of select="individual/first_name"/>
      </first_name>
      <last_name>
        <xsl:value-of select="individual/last_name"/>
      </last_name>
    </full_name>
  </xsl:template>
</xsl:transform>
```

# Using Multiple Templates in a Script

You can see processing instructions can get quite complex. It would be helpful if one set of processing instructions could call other sets of processing instructions. You can do this by splitting the processing instructions into templates that process sections of the input data.

## The XSLT script

Example 41 shows one way of separating the directives used in Example 40.

**Example 41:** *XSLT Script With Multiple Templates*

```
<?xml version="1.0"?>
<xsl:transform version="1.0"
               xmlns:ns1="http://www.iona.com/xslt"
               xmlns:out="http://www.iona.com/xslt"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <message>
      <transformer_reply>
        <xsl:apply-templates/>
      </transformer_reply>
    </message>
  </xsl:template>
  <xsl:template match="client_request">
    <xsl:apply-templates select="personal_details"/>
    <xsl:apply-templates select="individual"/>
  </xsl:template>
  <xsl:template match="personal_details">
    <xsl:attribute name="sex">
      <xsl:value-of select="@sex"/>
    </xsl:attribute>
  </xsl:template>
```

**Example 41:** *XSLT Script With Multiple Templates*

```
  <xsl:template match="individual">
    <full_name>
      <first_name>
        <xsl:value-of select="first_name"/>
      </first_name>
      <last_name>
        <xsl:value-of select="last_name"/>
      </last_name>
    </full_name>
  </xsl:template>
</xsl:transform>
```

CHAPTER 8

# Using Kerberos Security

*Kerberos security services are powerful and pervasive. Artix can easily accept Kerberos style security tokens.*

**Overview**

Artix services can use Kerberos tokens, embedded in WSSE headers, for authentication. Doing so requires that you:

- generate a keytab for iSF.
- configure your services to use iSF
- configure the iSF Kerberos adapter.

Artix clients can be coded to embed Kerberos tokens in WSSE headers. This requires using one of the one of the security APIs that provides access to the Kerberos KDC to get tokens. It also requires that your clients are deployed into an environment where running applications can initialize security contexts.

**Other resources**

For more information on the iSF and Kerberos see the Artix Security Guide.

**In this chapter**

This chapter discusses the following topics:

# Configuring Artix Services

**Overview**

Services developed using Artix do not require any code modifications to use Kerberos security. Security is set up through a combination of the following:

- a Kerberos keytab file
- the service's Artix configuration file
- the iSF's configuration
- the iSF Kerberos adapter's configuration file

**In this section**

This section discusses the following topics:

# Editing the Service's Configuration File

**Overview**

In order for an Artix service to take advantage of the iSF, the service needs to load the Artix security plug-in and the IIOP\TLS plug-in. The Artix security plug-in intercepts the WSSE headers and forwards them to the iSF server for authentication. The IIOP\TLS plug-in is required for communicating with the iSF server.

You also need to configure a number of properties that control the behavior of the Artix security plug-in. These properties tell the plug-in what type of security scheme to use, and at what level in the messaging chain security credentials are available.

**Modify the configuration**

To modify the configuration of an Artix service so that it uses Kerberos security do the following:

1. As the first line of your service's configuration add the following include statement:

```
include "ArtixInstallDir/etc/domains/artix-secure.cfg";
```

2. Locate the configuration scope for your service.
3. Add the following line before the service's `orb_plugins` entry:

```
plugins:artix_security:shlib_name = "it_security_plugin";
```

4. Add the value `security` to the beginning of the entry for `binding:artix:server_request_interceptor_list`.
5. Add the values `iiop_profile`, `giop`, `iiop_tls`, and `artix_security` to the entry for `orb_plugins`.
6. Set the value for `policies:asp:enable_authorization` to `false`.
7. Set the value for `plugins:asp:security_type` to `KERBEROS_TOKEN`.
8. Set the value for `plugin:asp:security_level` to `REQUEST_LEVEL`.
9. Set any other security related policies that you want to modify.

**Example**

Example 42 shows a configuration scope for a service that uses Kerberos security.

**Example 42:** *Service Configured to Use Kerberos Security*

```
server
{
  plugins:artix_security:shlib_name = "it_security_plugin";
  binding:artix:server_request_interceptor_list= "security";
  binding:client_binding_list = ["OTS+POA_Coloc", "POA_Coloc",
                                 "OTS+GIOP+IIOP", "GIOP+IIOP",
                                 "GIOP+IIOP_TLS"];
  orb_plugins = ["iiop_profile", "giop", "iiop_tls",
                 "artix_security"];
  policies:asp:enable_authorization = "false";
  plugins:asp:security_type = "KERBEROS_TOKEN";
  plugins:asp:security_level = "REQUEST_LEVEL";
  plugins:asp:authentication_cache_size = "5";
  plugins:asp:authentication_cache_timeout = "10";
};
```

# Configuring the iSF Server

**Overview**

The iSF server requires a certain amount of configuration information in order to run properly. There are a number of ways to specify the iSF server's configuration. One way is to place it in the global scope of the Artix configuration file. Another way is to create a sub-scope for the iSF server inside the configuration used for your application. However, the recommended way is to create an independent configuration scope for your iSF server instance and place the scope in an independent file.

**Create a configuration for the iSF server**

To create a configuration file containing the required information for running the iSF and to use the Kerberos security adapter do the following:

1. Create new file to hold your configuration.
2. As the first line of the configuration add the following include statement:

```
include "ArtixInstallDir/etc/domains/artix-secure.cfg";
```

3. Create a configuration scope for the iSF server.
4. From the `full_security` scope in the `artix-secure.cfg` configuration file, copy the configuration entry for `initial_references:IT_SecurityService:reference` into your iSF configuration scope.
5. Using the `full_security.security_service` scope as a template, set the configuration values for your security service instance.
6. Locate the entry for `plugins:java_server:system:properties`.
7. In this entry change `"is2.properties=iSFAdapterConfigDir/is2.properties.FILE"` to `"is2.properties=iSFAdapterConfigDir/is2.properties.KERBEROS"`.

**Example**

Example 43 shows a configuration scope for an instance of the iSF server that uses Kerberos security.

**Example 43:** *iSF Server Configuration*

```
iSF_service
{
  initial_references:IT_SecurityService:reference = "corbaloc:iiops:1.2@localhost:58483,
                                                it_iiops:1.2@localhost:58483/IT_SecurityService";

  password_retrieval_mechanism:inherit_from_parent = "true";
  principal_sponsor:use_principal_sponsor = "true";
  principal_sponsor:auth_method_id = "pkcs12_file";

  binding:client_binding_list = ["GIOP+EGMIOP", "OTS+TLS_Coloc+POA_Coloc", "TLS_Coloc+POA_Coloc",
                                 "OTS+POA_Coloc", "POA_Coloc", "GIOP+SHMIOP",
                               "CSI+OTS+GIOP+IIOP_TLS", "OTS+GIOP+IIOP_TLS", "CSI+GIOP+IIOP_TLS",
                                 "GIOP+IIOP_TLS", "CSI+OTS+GIOP+IIOP", "OTS+GIOP+IIOP",
                                 "CSI+GIOP+IIOP", "GIOP+IIOP"];

  policies:target_secure_invocation_policy:requires = ["Confidentiality"];
  policies:target_secure_invocation_policy:supports = ["Confidentiality",
                                                       "EstablishTrustInTarget",
                                                   "EstablishTrustInClient", "DetectMisordering",
                                                       "DetectReplay", "Integrity"];
  policies:client_secure_invocation_policy:requires = ["Confidentiality"];
  policies:client_secure_invocation_policy:supports = ["Confidentiality",
                                                       "EstablishTrustInTarget",
                                                   "EstablishTrustInClient", "DetectMisordering",
                                                       "DetectReplay", "Integrity"];
  policies:allow_unauthenticated_clients_policy = "true";

  orb_plugins = ["local_log_stream", "iiop_profile", "giop", "iiop_tls"];
  generic_server_plugin = "java_server";
  plugins:java_server:shlib_name = "it_java_server";
  plugins:java_server:class = "com.iona.corba.security.services.SecurityServer";
  plugins:java_server:classpath = "%{SECURITY_CLASSPATH}";
  plugins:java_server:jni_verbose = "false";

  plugins:java_server:X_options = ["rs"];
  plugins:security:direct_persistence = "true";
```

**Example 43:** *iSF Server Configuration*

```
  principal_sponsor:auth_method_data =
    ["filename=/artix/security/certificates/tls/x509/certs/services/administrator.p12",
     "password_file=/artix/security/certificates/tls/x509/certs/services/administrator.pwf"];
  plugins:java_server:system_properties =
    ["org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl",
     "org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.ORBSingleton",
     "is2.properties=/artix/security/security_service/is2.properties.KERBEROS",
     "java.endorsed.dirs=/artix/4.0/lib/endorsed"];
  plugins:local_log_stream:filename = "/artix/security_service/isf.log";

  policies:iiop:server_address_mode_policy:local_hostname = "localhost";
  plugins:security:iiop_tls:port = "58483";
  plugins:security:iiop_tls:host = "localhost";
  plugins:security:iiop:port = "58487";
  plugins:security:iiop:host = "localhost";
};
```

**More information**

For more information about Artix configuration see Configuring and Deploying Artix Solutions.

For more information about configuring the iSF server see the Artix Security Guide.

# Configuring the iSF Kerberos Adapter

**Overview**

The iSF uses adapters to communicate with the different types of authorization services. Each adapter requires a few properties to tell it what classes to use, where to look for the needed certificates, and the ports to use. The Kerberos adapter uses a file called `is2.properties.KERBEROS` to hold its properties. The location of the properties file is specified in the Artix configuration file and must be accessible by the iSF service.

In addition to a properties file, the Kerberos adapter also requires a JAAS configuration file. This file specifies the keytab file to use for authentication with the KDC. The default location for this file is the same directory as the iSF service is started from. You can change the location in the Kerberos adapter's properties file.

**Create a Kerberos adapter properties file**

To create a properties file for a Kerberos enabled iSF service do the following:

1. Copy *`ArtixInstallDir`*`\artix\4.0\etc\is2.properties.KERBEROS` to the location specified in your iSF service's Artix configuration file.
2. Open the new properties file in a text editor.
3. Change the value for `com.iona.isp.adapter.krb5.param.java.security.krb5.realm` to the name of the security realm under which the KDC is running.
4. Change the value for `com.iona.isp.adapter.krb5.param.java.security.krb5.kdc` to the IP address of the machine on which the KDC is running.
5. Change the value for `com.iona.isp.adapter.krb5.param.host.1` to the IP address of the machine on which the KDC is running.
6. Change the value for `com.iona.isp.adapter.krb5.param.PrincipalUserDN.1` to the username that was used to generate the keytab file used by the iSF for authenticating services.
7. Change the value for `com.iona.isp.adapter.krb5.param.PrincipalUserPassword.1` to the password associated with the username specified in step **6**.

8. Make any other modifications you require to fine tune your deployment.

**Example**

Example 44 shows an example of a Kerberos adapter properties file.

**Example 44:** *iSF Kerberos Properties File*

```
com.iona.isp.adapters=krb5

com.iona.isp.adapter.krb5.class=com.iona.security.is2adapter.krb5.IS2KerberosAdapter
com.iona.isp.adapter.krb5.param.java.security.krb5.realm=IONA.USA.COM
com.iona.isp.adapter.krb5.param.java.security.krb5.kdc=100.01.01.01
com.iona.isp.adapter.krb5.param.java.security.auth.login.config=jaas.conf
com.iona.isp.adapter.krb5.param.javax.security.auth.useSubjectCredsOnly=false

#To retrieve group infor from active directory, change this to true
com.iona.isp.adapter.krb5.param.RetrieveAuthInfo=false

com.iona.isp.adapter.krb5.param.host.1=100.01.01.01
com.iona.isp.adapter.krb5.param.port.1=389
com.iona.isp.adapter.krb5.param.PrincipalUserDN.1=iSF
com.iona.isp.adapter.krb5.param.PrincipalUserPassword.1=IONA
com.iona.isp.adapter.krb5.param.ConnectTimeout.1=15

com.iona.isp.adapter.krb5.param.UserNameAttr=CN
com.iona.isp.adapter.krb5.param.UserBaseDN=dc=boston,dc=amer,dc=iona,dc=com
com.iona.isp.adapter.krb5.param.version=3
com.iona.isp.adapter.krb5.param.UserObjectClass=Person
com.iona.isp.adapter.krb5.param.GroupObjectClass=group
com.iona.isp.adapter.krb5.param.GroupSearchScope=SUB
com.iona.isp.adapter.krb5.param.GroupBaseDN=dc=boston,dc=amer,dc=iona,dc=com
com.iona.isp.adapter.krb5.param.GroupNameAttr=CN
com.iona.isp.adapter.krb5.param.MemberDNAttr=memberOf
com.iona.isp.adapter.krb5.param.MaxConnectionPoolSize=1
com.iona.isp.adapter.krb5.param.MinConnectionPoolSize=1

## Single Sign On Session Info
is2.sso.session.timeout=600
is2.sso.session.idle.timeout=60
is2.sso.cache.size=200

## AuthorizationManager configuration
com.iona.security.azmgr.adminUserName=IONAAdmin
com.iona.security.azmgr.PersistencePropertyFileName=persistent.PROPERTIES
```

**Creating the JAAS configuration**

To create a JAAS configuration file do the following:

1. Using any text editor create a new file called `jaas.properties`.
2. Copy the following text into the file.

```
com.sun.security.jgss.initiate {
com.sun.security.auth.module.Krb5LoginModule required
principal="iSF@IONA.USA.COM"
useKeyTab = true keyTab="isf.keytab";
};

com.sun.security.jgss.accept {
com.sun.security.auth.module.Krb5LoginModule required
storeKey=true principal="iSF@IONA.USA.COM"
useKeyTab = true keyTab="isf.keytab";
};
```

3. Change the value for both `principal` entries to the fully qualified username used to connect to the KDC.
4. Change the value of both `keyTab` entries to the location of the keytab file used for authorizing service requests.

**More information**

For more information about configuring the iSF Kerberos adapter see the Artix Security Guide.

# Modifying Artix Consumers

**Overview**

Unlike Artix services, Artix consumers require code level modification to use Kerberos security. Consumers need to request Kerberos tokens from the KDC before making requests on a service. They also need to package the Kerberos token into a WS-Security compliant header in all request made against a secure service.

**In this section**

This section discusses the following topics:

# Getting a Kerberos Token

**Overview**

There are a number of security toolkits available that allow you to contact a KDC and request tokens. For example, Sun provides the Java Generic Security Service with its JDKs. MIT provides Kerberos APIs for a number of platforms. There is a GNU implementation of the GSS for C++. Microsoft also provides APIs for working with KDCs. Which APIs you choose to use depends on the language in which you wish to develop your clients. You must also decide which platforms you will use.

**Example**

Example 45 shows sample Java code that uses JGSS APIs to get a token.

**Example 45:** *Getting a Kerberos Token in Java*

```
import java.io.*;
import java.util.Properties;
import java.security.cert.*;
import sun.misc.BASE64Encoder;
import org.ietf.jgss.GSSContext;
import org.ietf.jgss.GSSName;
import org.ietf.jgss.GSSManager;
import org.ietf.jgss.Oid;

Oid krb5Oid = new Oid("1.2.840.113554.1.2.2");
GSSManager manager = GSSManager.getInstance();
GSSName serverName = manager.createName(service, null);
GSSContext context = manager.createContext (serverName, krb5Oid, null,
   GSSContext.INDEFINITE_LIFETIME);
context.requestMutualAuth(true);
context.requestConf(true);
context.requestInteg(true);
byte[] token = new byte[0];
token = context.initSecContext(token, 0, token.length);
context.dispose();
```

# Adding the Kerberos Token to a Request

**Overview**

In order for an Artix service to use a Kerberos token to authenticate a request, it must be packed into a WS-Security header. The WS-Security header is sent along with each request a client makes. Artix uses its context mechanism for placing Kerberos tokens into WS-Security headers and ensuring that they are sent out with each request.

This procedure has three steps:

1. Encode the Kerberos token into a Base 64 string.
2. Get the Artix bus security context.
3. Place the encoded Kerberos token into the bus security context.

**Using C++**

Embedding a Kerberos token into the SOAP header of a request using the Artix C++ APIs requires you to do the following:

1. Make sure that your application makefile is configured to link with the `it_context_attribute` library (`it_context_attribute.lib` on Windows and `it_context_attribute.so` or `it_context_attribute.a` on UNIX) which contains the bus-security context stub code.
2. Get a reference to the current `IT_ContextAttributes::BusSecurity` context data type, using the Artix context API (see Developing Artix Applications in C++).
3. Set the `WSSEKerberosv5SToken` property on the `BusSecurity` context using the `setWSSEKerberosv5SToken()` method.

Example 46 shows how to set the Kerberos token prior to invoking a remote operation.

**Example 46:** *Embedding a Kerberos Token Using C++*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>
// Include header files related to the bus-security context
#include <it_bus_pdk/context.h>
#include <it_bus_pdk/context_attrs/bus_security_xsdTypes.h>
```

**Example 46:** *Embedding a Kerberos Token Using C++*

```
IT_USING_NAMESPACE_STD
using namespace IT_ContextAttributes;
using namespace IT_Bus;

int main(int argc, char* argv[])
{
  try
  {
    IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);
    ContextRegistry* context_registry =
                    bus->get_context_registry();

    // Obtain a reference to the ContextCurrent
    ContextCurrent& context_current =
                   context_registry->get_current();

    // Obtain a pointer to the Request ContextContainer
    ContextContainer* context_container =
                     context_current.request_contexts();

1   // Obtain a reference to the context
   AnyType* info = context_container->get_context(
              IT_ContextAttributes::SECURITY_SERVER_CONTEXT,
              true);

2   // Cast the context into a BusSecurity object
    BusSecurity* bus_security_ctx =
                  dynamic_cast<BusSecurity*> (info);

3   // Set the Kerberos token
    bus_security_ctx->setWSSEKerberosv5SToken(
      kerberos_token_string);
    ...
  }
  catch(IT_Bus::Exception& e)
  {
    cout << endl << "Error : Unexpected error occured!"
         << endl << e.message() << endl;
    return -1;
  }

  return 0;
}
```

The preceding code can be explained as follows:

1. The `IT_Bus::ContextContainer::get_context()` function is called with its second parameter set to true, indicating that a context with that name will be created if none already exists.
2. Cast the `IT_Bus::AnyType` instance, `info`, to its derived type, `IT_ContextAttributes::BusSecurity`, which is the bus-security context data type.
3. Use the `BusSecurity` API to set the WSSE Kerberos token, `kerberos_token_string`. The argument to `setWSSEKerberosv5SToken()` is a base-64 encoded Kerberos token received from a Kerberos server.

The next operation invoked from this thread will include the specified Kerberos token in the request message.

---

**Using Java**

Embedding a Kerberos token into the SOAP header of a request in Artix Java requires you to do the following:

1. Create a new `com.iona.schemas.bus.security_context.BusSecurity` context data object.
2. Set the `WSSEKerberosv2SToken` property on the BusSecurity context using the `setWSSEKerberosv2SToken()` method.
3. Set the bus-security context for the outgoing request message by calling `setRequestContext()` on an `IonaMessageContext` object (see Developing Artix Applications in Java).

Example 47 shows how to set the Kerberos token in a Java client prior to invoking a remote operation.

**Example 47:** *Embedding a Kerberos Token Using Java*

```
// Java
import javax.xml.namespace.QName;
import javax.xml.rpc.*;
import com.iona.jbus.Bus;
import com.iona.jbus.ContextRegistry;
import com.iona.jbus.IonaMessageContext;
import com.iona.schemas.bus.security_context.BusSecurity;
import com.iona.schemas.bus.security_context.BusSecurityLevel;
...
```

**Example 47:** *Embedding a Kerberos Token Using Java*

```
// Set the BuSecurity Context
//---------------------------
// Insert the following lines of code prior to making a
// WS-secured invocation:
1 BusSecurity security = new BusSecurity();

2 security.setWSSEKerberosv5SToken(kerberos_token_string);

3 QName SECURITY_CONTEXT = new QName(
                "http://schemas.iona.com/bus/security_context",
                "bus-security");

4 ContextRegistry registry = bus.getContextRegistry();

5 IonaMessageContext contextimpl =
    (IonaMessageContext)registry.getCurrent();

6 contextimpl.setRequestContext(SECURITY_CONTEXT, security);
```

The preceding code can be explained as follows:

1. Create a new `com.iona.schemas.bus.security_context.BusSecurity` object to hold the context data and initialize the `WSSEKerberosv2SToken` on this BusSecurity object.
2. Use `BusSecurity.setWSSEKerberosv5SToken()` to place the base-64 encoded Kerberos token received from a Kerberos server.
3. Initialize the name of the bus-security context. Because the bus-security context type is pre-registered by the Artix runtime (thus fixing the context name) the bus-security name must be set to the value shown here.
4. The `com.iona.jbus.ContextRegistry` object manages all of the context objects for the application.
5. The `com.iona.jbus.IonaMessageContext` object returned from `getCurrent()` holds all of the context data objects associated with the current thread.
6. Call `setRequestContext()` to initialize the bus-security context for outgoing request messages.

The next operation invoked from this thread will include the specified Kerberos token in the request message.

CHAPTER 9

# Using Artix Security with Non-Artix Clients

*The Artix iSF uses a security token based on open standards for authentication and can be used with any Web services client that can pass a valid token.*

## Overview

Interoperability is one of the major features of a service-oriented approach to software design. Artix ensures interoperabilty on the security front by using WS-Security tokens for authentication. The WS-Security standard (http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf) specifies a token that is packaged in a SOAP header. A secure Artix server will accept a valid WS-Security token from any source and use it to authenticate requests.

## WS-Security tokens

The XMLSchema definition for the WS-Security token can be found at http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd. The token can contain the following types of tokens:

- username/password
- base 64 binary encoded Kerberos tickets
- base 64 binary encoded X.509 certificates

- URI to security tokens stored at a remote location

### Adding the token to requests

The WS-Security token must be added as a SOAP header on all requests that are made on a secure Artix service. In order for an Artix service to use the token the header must be formatted so that:

1. The namespace for the token definition is specified as `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd`.
2. The header element is named `wsse:Security`.
3. The elements used inside the WS-Security token are supported by Artix. For more information on the types of tokens supported by Artix see the Artix Security Guide.

Example 48 shows the required format of the WS-Security token SOAP header.

**Example 48:** *WS-Security token SOAP Header*

```
<?xml version="1.0" encoding="utf-8"?>
<S11:Envelope xmlns:S11="..."
 xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <S11:Header>
    <wsse:Security
   xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd
   ">
      ...
    </wsse:Security>
  </S11:Header>
  <S11:Body wsu:Id="MsgBody">
    ...
  </S11:Body>
</S11:Envelope>
```

The process by which you add a SOAP header to a message depends on the Web service platform you are using.

### Example

To create an Axis client that makes requests on a secure Artix service is fairly straightforward. Axis implements the JAX-RPC standard and uses the SAAJ APIs for manipulating SOAP messages. This example shows code for adding a username/password WS-Security token using an Axis client.

Because the WS-Security token needs to be attached to every request sent to a secure Artix service, it is most effective to implement the code for adding the token to the SOAP head as a `Handler` that is added to the client's `Hander` chain. Example 49 shows the code for implementing such a `Handler`.

**Example 49:** *WS-Security Handler*

```
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.Name;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;

public class WSSEUsernamePasswordHandler implements Handler
{
1   private static final String WSSE_URI =
     "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
     curity-secext-1.0.xsd";

    ...
    public boolean handleRequest(MessageContext context)
    {
2     String username = (String)
      context.getProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY);
       String password = (String)
      context.getProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY);

       try
       {
3        SOAPMessageContext smc = (SOAPMessageContext)context;
4        SOAPMessage msg = smc.getMessage();
5        SOAPPart sp = msg.getSOAPPart();
6        SOAPEnvelope se = sp.getEnvelope();
7        SOAPHeader sh = se.getHeader();

8        Name name = se.createName("Security", "wsse", WSSE_URI);
9        SOAPElement wsseSecurity = sh.addChildElement(name);
```

**Example 49:** *WS-Security Handler*

```
10        SOAPElement usernameToken =
        wsseSecurity.addChildElement("UsernameToken", "wsse");

11        SOAPElement usernameElement =
        usernameToken.addChildElement("Username", "wsse");
12        usernameElement.addTextNode(username);

13        SOAPElement passwordElement =
        usernameToken.addChildElement("Password", "wsse");
14        passwordElement.addTextNode(password);
       }
       catch (SOAPException e)
       {
         System.out.println(e);
       }

       return true;
     }
     ...
}
```

The code in Example 49 does the following:

1. Set a private variable to the WS-Security token's namespace.
2. Get the username and password from the `MessageContext`.

   **Note:** In this example, the username and password are stored using `Stub` properties.

3. Cast the `MessageContext` into a `SOAPMessgeContext`.
4. Get the `SOAPMessage` from the `SOAPMessageContext`.
5. Get the `SOAPPart` from the `SOAPMessage`.
6. Get the `SOAPEnvelope` from the `SOAPPart`.
7. Get the `SOAPHeader` from the `SOAPEnvelope`.
8. Create the name for your the token's SOAP header element using `SOAPEnvelope.createName()`.

   **Note:** The name of the SOAP header element should result in `wsse:Security` and have the proper namespace declaration.

9. Add a new `SOAPElement` to the SOAP message header using `SOAPHeader.addChildElement()`.
10. Add a `wsse:UsernameToken` child element to the `SOAPElement` added to the SOAP header.
11. Add a `wsse:Username` child element to the `wsse:UsernameToken` element.
12. Set the value of the `wsse:Username` element to the username retrieved from the `Stub` properties.
13. Add a `wsse:Password` child element to the `wsse:UsernameToken` element.
14. Set the value of the `wsse:Password` element to the password retrieved from the `Stub` properties.

Example 50 shows the client code for setting the username and password on the `Stub`.

**Example 50:** *Client Setting Username and Password*

```
// Java
CISOAPStub = (CISOAPStub) new
   CI_ServiceLocator().getSOAPOverHTTPDocLiteral();

CISOAPStub._setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY,
                        "Adie");
CISOAPStub._setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY,
                        "Baby");
```

CHAPTER 10

# Using Unmapped SOAP Message Elements

*You can place XML content that is not mapped into an Artix generated class into your SOAP messages.*

**Overview**

You may want to include XML elements that do not have a corresponding Artix generated class in a SOAP message. For example, your application deals directly with XML data and you do not want the overhead of converting the XML data into a Java object and then converting back into XML in to be inserted into a SOAP message. This can be done using and `xsd:any` type.

**In this chapter**

This chapter discusses the following topics:

# Unmapped XML Data and xsd:any

**Overview**

The `xsd:any` type is an XMLSchema element that defines elements that can contain undefined XML data. The JAX-RPC specification dictates that the `xsd:any` type be represented by an instance of a class that implements the `javax.xml.soap.SOAPElement` interface. The `SOAPElement` interface implements the `org.w3c.dom.Node` interface and offers a similar API as an object implementing the `org.w3c.dom.Document` interface for manipulating its content.

Therefore, if you want to use XML data that is not defined in an Artix contract you can do the following:

1. Manipulate the data in a document object.
2. Use the document to populate an instance of an object implementing `SOAPElement`
3. Pass the data in an operation that uses an `xsd:any` as a parameter.

You can find a nice introduction to this topic at http://www-106.ibm.com/developerworks/library/ws-xsdany.html.

**Using xsd:any in a contract**

The contract fragment shown in Example 51 illustrates how the `xsd:any` type can be used within a message part.

**Example 51:** *xsd:any in a Contract*

```
<types>
  <schema targetNamespace="http://www.iona.com/artix/wsdl"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="unmappedType">
      <sequence>
        <xsd:any namespace="##other"
                 processContents="skip"/>
      </sequence>
    </complexType>
    <element name="request" type="tns:requestType"/>
  </schema>
</types>
```

**Example 51:** *xsd:any in a Contract*

```
<message name="sayHiRequest">
  <part element="tns:request" name="request"/>
</message>
```

`unmappedType` is declared with a single element of `xsd:any`. When converted into Java code, the generated `UnmappedType` class, derived from the `unmappedType` type definition, will have a member variable of type `SOAPElement`, called `_any`, to represents the xsd:any element in the contract definition as shown in Example 52.

**Example 52:** *Generated Class Using xsd:any*

```
import javax.xml.soap.SOAPElement;

public class UnmappedType
{
    private SOAPElement _any;

    public SOAPElement get_any() {
        return _any;
    }

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

An operation that uses the `sayHiRequest` message will have a parameter of type `UnmappedType`.

---

**Code for working with raw XML**

You have several approaches obtaining the XML data that represents your unmapped message content. The application might create a document object directly, or it might initialize a document object from a file containing XML content. How you manipulate the content of the document object is managed through the DOM APIs.

Once you have a document object, you must transfer its contents into a `SOAPElement` instance. Unfortunately neither the DOM nor the `SOAPElement` interface defines an API that will perform this task. You must handle this as part of your application.

Fortunately, Artix includes a class, `com.iona.webservices.saaj.util.SAAJUtils`, that you can use to transfer content between DOM and `SOAPElement` objects. As shown in Example 53, this class will also create and populate a `SOAPElement` instance with the content obtained from an XML file.

**Example 53:** *Creating a SOAPElement from an XML File*

```
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPFactory;
import java.io.File;
import java.io.FileInputStream;
import org.xml.sax.InputSource;
import com.iona.webservices.saaj.util.SAAJUtils;

File file = new File("Test.xml");
SOAPElement element = null;

InputSource source = new InputSource(new FileInputStream(file));
SOAPFactory factory = SOAPFactory.newInstance();
element = SAAJUtils.parseSOAPElement(factory, source);
```

# When Only One Side Uses Unmapped XML Data

**Overview**

If only one side of an application is implemented to use unmapped XML data, Artix can handle it. For example, your client application might work with raw XML data and sends as unmapped XML. However, your server is written to expect mapped data that Artix can convert into a Java object that represents the elements in the message. For Artix to handle this situation, you must write separate contracts for the client and the server applications. The client-side contract will represent the message content as unmapped data while the server-side contract will provide a type mapping that corresponds to the unmapped message content.

This section will examine an example of such a usecase.

**The XML File**

A file, `employee.xml`, includes an employee record. The client application will use the `SAAJUtils` class to convert the content of this file into a `SOAPElement` and then send the content of the `SOAPElement` as the message.

The file includes the content shown in Example 54.

**Example 54:** *XML Content*

```
<?xml version='1.0' encoding='utf-8'?>
<employee><fname>Henry</fname><lname>James</lname></employee>
```

**Client contract**

In this simple example, the client's contract contains two type definitions:

- a complex type that contains the xsd:any.
- an element that wraps the complex type.

The target namespace and the type names used in the client's contract must match the names used in the server's contract. If they are different the messages will not be marshalled properly.

The rest of the client's contract contains simple messages, a SOAP (doc/literal) binding, and a service/port definition.

Example 55 shows a contract for a client that uses unmapped XML data.

**Example 55:** *Contract for a Client Sending Unmapped XML Data*

```
<definitions name="soapelement.wsdl"
             targetNamespace="http://www.iona.com/artix/wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://www.iona.com/artix/wsdl"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/artix/wsdl"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="requestType">
      <sequence>
        <any namespace="##other" processContents="skip"/>
      </sequence>
    </complexType>
    <element name="request" type="tns:requestType"/>
  </schema>
</types>
<message name="sayHiResponse"/>
<message name="sayHiRequest">
  <part element="tns:request" name="request"/>
</message>
<portType name="Greeter">
  <operation name="sayHi">
    <input message="tns:sayHiRequest" name="sayHiRequest"/>
    <output message="tns:sayHiResponse" name="sayHiResponse"/>
  </operation>
</portType>
<binding name="Greeter_SOAPBinding" type="tns: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>
</binding>
```

**Example 55:** *Contract for a Client Sending Unmapped XML Data*

```
<service name="SOAPService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location="http://localhost:9000"/>
  </port>
</service>
</definitions>
```

**Client application**

After you generate Java from the contract, the client application includes three files whose names are derived from the name assigned to the port type, and a fourth file corresponding to the complex type `requestType`. Since the `xsd:any` wrapped by `requestType` will be implemented by an instance of `SOAPElement`, no code is generated for the type wrapped within `RequestType`. You need to add code to the file containing the client mainline.

The method `sayHi()` requires an argument of type `RequestType`. This argument must be initialized with a `SOAPElement` instance that contains the content of the file `employee.xml`. You can create the `SOAPElement` instance by using the code in Example 53. Just be certain to provide the correct path to `employee.xml`.

In the mainline code, where `sayHi()` is called, you will need to set the `SOAPElement` instance into the method argument as shown in Example 56.

**Example 56:** *Setting the SOAPElement Instance*

```
RequestType request = new RequestType();
request.set_any(element);
impl.sayHi(_request);
```

That's all there is to sending unmapped data as a SOAP message, although in a more complex application, your client code might would be responsible for transferring data from a document into the `SOAPElement`.

**Server contract**

In writing this file, it is essential that the target namespaces in both the opening definitions and schema tags be the same as the corresponding target namespaces assigned in the client's contract. The namespace is included in the message content and if the client and server processes are not working with the same namespace, the message will not be compatible across the applications.

The types section of this WSDL file includes three type definitions:

- an XMLSchema description of the data in `employee.xml`.
- a redefinition of `requestType` using the new complex type in place of the `xsd:any` type.
- a duplicate of the element used to wrap the unmapped XML data in the client's contract.

The remainder of contract is virtually identical to the client's contract as shown in Example 57.

**Example 57:** *Server Contract*

```
<definitions name="soap"
             targetNamespace="http://www.iona.com/artix/wsdl"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xsd1="http://www.iona.com/artix/wsdl"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/artix/wsdl"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <complexType name="employee">
    <sequence>
      <element maxOccurs="1" minOccurs="1" name="fname"
               type="xsd:string"/>
      <element maxOccurs="1" minOccurs="1" name="lname"
               type="xsd:string"/>
    </sequence>
  </complexType>
  <complexType name="requestType">
  <sequence>
    <element maxOccurs="1" minOccurs="1" ref="xsd1:employee"/>
  </sequence>
  </complexType>
  <element name="request" type="xsd1:requestType"/>
  </schema>
</types>
```

**Example 57:** *Server Contract*

```
<message name="sayHiResponse"/>
<message name="sayHiRequest">
  <part element="xsd1:request" name="request"/>
</message>
<portType name="Greeter">
  <operation name="sayHi">
    <input message="xsd1:sayHiRequest" name="sayHiRequest"/>
    <output message="xsd1:sayHiResponse" name="sayHiResponse"/>
  </operation>
</portType>
<binding name="Greeter_SOAPBinding" type="xsd1: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>
</binding>
<service name="SOAPService">
  <port binding="xsd1:Greeter_SOAPBinding" name="SoapPort">
    <soap:address location="http://localhost:9000"/>
  </port>
</service>
</definitions>
```

**Server application**

After you generate Java code from the server's contract, the server application will include four files whose names are derived from the name assigned to the port type, and two files corresponding to the complex types `employee` and `requestType`. You need to add code to the file containing the servant implementation class.

The parameter to the `sayHi()` method is an instance of `RequestType` as it was in the client. However, `RequestType` does not contain a member variable of `SOAPElement`. It contains a member variable of `Employee`. `Employee` is the class generated using the XMLSchema description of the

XML data. When you extract the encapsulated Employee object using the code show in Example 58, its contents should match the contents of the original `employee.xml` file.

**Example 58:** *Getting the Employee Object*

```
Employee e = request.getEmployee();
```

# Index