Artix™

Technical Use Cases

Version 3.0, October 2005

Making Software Work Together™

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, Orbix Mainframe, Orbix Connect, Artix, Artix Mainframe, Artix Mainframe Developer, Mobile Orchestrator, Orbix/E, Orbacus, Enterprise Integrator, Adaptive Runtime Technology, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice.

Copyright © 1999-2005 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this publication are covered by the trademarks, service marks, or product names as designated by the companies that market those products.

Updated: 11-Nov-2005

Contents

Preface vii
- What is Covered in this Book vii
- Who Should Read this Book vii
- Finding Your Way Around the Library vii
- Searching the Artix Library ix
- Online Help ix
- Additional Resources x
- Document Conventions x

Chapter 1 Building a Client for a Web Service 1
- Importing WSDL into Artix 2
- Building the Client 4
  - Building a C++ Client 5
  - Building a Java Client 9
- Configuring Artix to Deploy the Client 14

Chapter 2 Building a Web Service 15
- Defining a Service in WSDL 16
  - Creating a WSDL Contract 17
  - Defining the Data Used by the Service 18
  - Defining the Messages Used by the Service 22
  - Defining the Service's Interface 23
  - Defining the Payload Format 25
  - Defining the Service's Endpoint Information 26
- Building the Service 27
  - Building a C++ Service 28
  - Building a Java Service 30
- Configuring Artix to Deploy the Service 33

Chapter 3 Web Service Enabling Backend Services 35
- Describing a Service in WSDL 37
  - Starting with CORBA IDL 38
  - Starting with a COBOL Copybook 40

Starting with a Java Class 43
Defining the Service's Endpoint Information 45
Defining a SOAP/HTTP Endpoint 49
Configuring and Deploying a Switching Service 52

Chapter 4 Using Artix with .NET 55
Building a .NET Client for an Artix Service 56
Building an Artix Client for a .NET Service 59
Using Artix to Bridge from a Backend Service to .NET 61
Using Artix Services from .NET clients 63
Using the Artix Locator 64
Using the Artix Session Manager 68

Chapter 5 Using Artix with WebSphere MQ 75
Artix and MQ on a Single Computer 77
Client and Server on Different MQ Servers 78
Client on MQ Client and Server on MQ Server 81
Using a Remote MQ Server 83
Using a Remote MQ Server from Full MQ Installations 85

Chapter 6 Writing XSLT Scripts for the Artix Transformer 87
The XSLT Script Template 89
Transforming a Sequence into a String 91
Modifying a Simple Sequence 93
Working with Nested Input Sequences 96
Working with Attributes in an Input Message 99
Working with Attributes in the Output Message 102
Working with Nested Sequences in an Output Message 105
Using Multiple Templates in a Script 108

Chapter 7 Using Artix Security with Non-Artix Clients 111

Chapter 8 Using Unmapped SOAP Message Elements 117
Unmapped XML Data and xsd:any 118
When Only One Side Uses Unmapped XML Data 121

Glossary 127

Index **137**

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.

Finding Your Way Around the Library

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

If you are new to Artix

You may be interested in reading:

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.

- Getting Started with Artix describes basic Artix and WSDL concepts.

To design and develop Artix solutions

Read one or more of the following:

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

To configure and manage your Artix solution

Read one or more of the following:

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

Reference material

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

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

Have you got the latest version?

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

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

Searching the Artix Library

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

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

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

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

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

Online Help

Artix Designer includes comprehensive online help, providing:

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

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

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

Additional Resources

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

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

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

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

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

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

Keying Conventions

This book uses the following keying conventions:

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

`...`
`.`
`.`
`.`
	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

`[]`	Brackets enclose optional items in format and syntax descriptions.
`{}`	Braces enclose a list from which you must choose an item in format and syntax descriptions.
`|`	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in `{}` (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select **File|Open**).

CHAPTER 1

Building a Client for a Web Service

Artix makes building a Web service client simple.

Overview

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

Building a Web service client with Artix involves three steps:

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

In this chapter

This chapter discusses the following topics:

Importing WSDL into Artix	page 2
Building the Client	page 4
Configuring Artix to Deploy the Client	page 14

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 client. 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|Iona Artix 3.0|Iona Artix 3.0|Artix Designer**. You can also use the `start_eclipse` script located under *InstallDir*`\artix\3.0\eclipse`.

On Linux platforms, start the Designer using the supplied `start_eclipse` script. The script is located in *InstallDir*`/artix/3.0/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 client

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|Other** from the Eclipse tool bar to open the **New** wizard.
3. Under **IONA Artix Designer** select **WSDL File**.
4. Click **Next**.
5. From the **WSDL File** window click **Advanced** to open the file selection panel.

6. Select **Link to file in the file system** to make the file selection box available.
7. Click **Browse** to open a file selection dialog.
8. Browse to the location of the contract you want to import and select it.
9. Enter a name for the contract in the **File name** box.
10. 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 Client

Overview

Using a WSDL contract, the Artix code generation tools generate most of the code you need to implement a Web service client. 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 client 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 client 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++ Client	page 5
Building a Java Client	page 9

Building a C++ Client

Overview

Building a client using the Artix C++ APIs is a four step process. First you generate the stub code from the WSDL contract. Using the generated code as a base, you add the code for initializing the Artix bus and a service proxy. Then you add the business logic to your client. Once the client is fully coded, you need to make the C++ application.

Generating code from Artix Designer

To generate C++ code for a Web service client from within the Designer, do the following:

1. Create a new code generation profile for you client application.
2. Fill in the information for your project on the **General** tab.
3. Select the **Generation** tab to bring up the code generation options panel.
4. Under **Generation Type**, select **Application**.
5. Under **Application Type**, select **Client**.
6. Under **Development Language**, select **C++**.
7. Select any services you want to use under **Optional Services**.
8. Enter a **C++ Namespace** and a **C++ Declaration Specification**.
9. Select **Generate a makefile** if you want to create a platform specific makefile.
10. Click **Apply** to save the settings.
11. Select the **WSDL Details** tab to bring up the WSDL service details panel.
12. From the table, select the service/port combination for which you want to generate code.
13. To generate a sample client click in the **Sample** column to change the **No** to **Yes**.
14. To generate code click **Run**.

The generated code is located under the **outputs\applications** folder of your project. The generated code, along with the Artix configuration file and start scripts for the application, is placed in a folder with the name you entered for the code generation profile. Underneath the application's root directory, your C++ code is placed into the **src** folder.

Generating code from the command line

To generate Artix C++ code for a Web service client, 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 `wsdltocpp` as shown below.

```
wsdltocpp -client -sample -m NAKE wsdl_file.wsdl
```

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

   > **Note:** If you are on a UNIX system substitute `MAKE` for `NMAKE`. This will generate a UNIX make file.

For more information on the `wsdltocpp` tool see Developing Artix Applications in C++.

Initializing the client

To create a C++ `main()` for an Artix client, 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 proxy using one of the three provided constructors:
   - *portTypeName*`Client()` instantiates a default client proxy that uses the first `<service>` and `<port>` listed in the application's contract. The contract must reside in the same directory from which the application is run.
   - *portTypeName*`Client(`*wsdlPath*`)` instantiates a default client proxy using the contract specified by *wsdlPath*.
   - *portTypeName*`Client(`*wsdlPath*`,` *service*`,` *port*`)` instantiates a client proxy using the service description specified by the combination of *service* and *port* from the specified contract.

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

Example 1: *Started C++ Client 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 client's business logic

Once the Artix bus is initialized and the proxy is created, all that remains to create a fully functional Web service client is the client's business logic. The code for the client'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 Client

Overview

Building a client using the Artix Java APIs is a four step process. First you generate the stub code from the WSDL contract. Using the generated code as a base, you add the code for initializing the Artix bus and a service proxy. Then you add the business logic to your client. Once the client is fully coded, you need to build the Java application.

Generating code from the Designer

Generating Java code from within Artix Designer allows you to have a unified environment for editing your contract, editing your code, and debugging your application. It will not overwrite changes to your implementation code if you need to regenerate your stubs.

To generate Java code for a Web service client from within the Designer, do the following:

1. Create a new code generation profile for you client application.
2. Fill in the information for your project on the **General** tab.
3. Select the **Generation** tab to bring up the code generation options panel.
4. Under **Generation Type**, select **Application**.
5. Under **Application Type**, select **Client**.
6. Under **Development Language**, select **Java**.
7. Select any services you want to deploy under **Optional Services**.
8. Deselect **Override namespace as package name** if you want to use the namespaces from the contract as the package names.
9. Click **Apply**.
10. Select the **WSDL Details** tab to bring up the WSDL service details panel.
11. From the table, select the service/port combination for which you want to generate code.
12. To generate a sample client click in the **Sample** column to change the **No** to **Yes**.
13. To generate code click **Run**.

When you are returned to the main Designer screen, you will notice that two new items have been added to your project's resource tree. The **bin** folder contains the compiled Java classes for your newly generated application. The **.classpath** file is used by the Eclipse framework to control the classpath used by the project at runtime.

The generated code is located under the **outputs\applications** folder of your project. The generated code, along with the Artix configuration file and start scripts for the application, is placed in a folder with the name you entered for the code generation profile. Underneath the application's root directory, your Java code is placed into the **src** folder and is placed into folders that reflect the package structure of the classes.

Generating code from the command line

To generate Artix Java code for a Web service client, 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 `wsdltojava` as shown below.

```
wsdltojava -client -ant 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 client, 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 javatowsdl tool, see Developing Artix Applications in Java.

Initializing the client

To create a Java `main()` for an Artix client 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 client'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 client'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 client's business logic

Once the Artix bus is initialized and the service proxy is created, all that remains to create a fully functional Web service client is the client's business logic. The code for the client's 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 client 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 Artix Applications with Java.

Configuring Artix to Deploy the Client

Overview

Once your client 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 client 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 clients 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 Deploying and Managing 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 Web Service

Artix makes building a Web service simple.

Overview

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

Building a Web service with Artix involves three steps:

1. Defining your service 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	page 16
Building the Service	page 27
Configuring Artix to Deploy the Service	page 33

Defining a Service in WSDL

Overview

The first step in building a Web service is getting a WSDL contract that defines the service 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 contract. 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	page 17
Defining the Data Used by the Service	page 18
Defining the Messages Used by the Service	page 22
Defining the Service's Interface	page 23
Defining the Payload Format	page 25
Defining the Service's Endpoint Information	page 26

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|Iona Artix 3.0|Iona Artix 3.0|Artix Designer**. You can also use the `start_eclipse` script located under *`InstallDir`*`\artix\3.0\eclipse\`.

On Linux platforms, start the Designer using the supplied `start_eclipse` script. The script is located in *`InstallDir`*`/artix/3.0/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 Web service. 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.
2. Using the Eclipse new resource wizard, select **WSDL File** from the **IONA Artix Designer** folder.
3. Click **Next>**.
4. Enter a new for the new contract in the **File Name** field.
5. Click **Finish**.

Defining the Data Used by the Service

Overview

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

For more information on defining types, see Designing Artix Solutions.

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 XMLSchema.
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 **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 XMLSchema.

   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. Select the type of the element you want to add to the structure from the **Type** drop-down list.
11. Enter a name for the new element.
12. Specify the desired attributes for the element.
13. Click **Add** to save the element to the **Element List** table.
14. If you need to edit an element definition:
    i. Select the element from the **Element List** table.
    ii. Make any changes you desire.
    iii. Click **Update** to save the changes.
15. Repeat steps 10 through 14 until you have finished adding elements to the structure.
16. Click **Next>**.

17. Select the type of the attribute you want to add to the structure from the **Type** drop-down list.
18. Enter a name for the new attribute.
19. If the attribute must be present in every instance of this type, place a check in the **Required** check box.
20. Click **Add** to save the attribute to the **Attribute List** table.
21. If you need to edit an attribute definition:
    i. Select the attribute from the **Attribute List** table.
    ii. Make any changes you desire.
    iii. Click **Update** to save the changes.
22. Repeat steps **17** through **21** until you have finished adding attributes to the structure.
23. 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 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 XMLSchema.

   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:

- substitution groups
- 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 Designing Artix Solutions.

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 XMLSchema 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. Enter a name for the message part.
8. Select a data type from the **Type** drop-down list.
9. Click **Add** to save the new part to the **Part List** table.
10. If you need to edit a part definition:
    i. Select the part from the **Part List** table.
    ii. Make any changes you desire.
    iii. Click **Update** to save the changes.
11. Repeat steps **7** through **10** until you have finished adding parts to the message.
12. 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 Designing Artix Solutions.

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. Select a message type for the new operation message from the **Type** drop-down list.
11. Select the global message that defines the data passed by this operation message from the **Message** drop-down list.
12. Enter a name for the operation message.
13. Click **Add** to add the message to the **Operation Messages** table.
14. If you need to edit an operation message:
    i. Select the part from the **Operation Messages** table.
    ii. Make any changes you desire.

iii. Click **Update** to save the changes.

15. Repeat steps **10** through **14** until all of the operational messages have been specified.
16. 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. Select a message type for the new operation message from the **Type** drop-down list.
7. Select the message that defines the data passed by this operation message from the **Message** drop-down list.
8. Enter a name for the operation message.
9. Click **Add** to add the message to the **Operation Messages** table.
10. Repeat steps **10** through **14** until all of the operational messages have been specified.
11. 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 services typically use SOAP as the payload format.

For more information on defining a payload format see Designing Artix Solutions.

Adding a SOAP binding using the designer

To add a SOAP binding do the following:

1. Alt-click the **Bindings**.
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**.
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. Click **Finish**.

Defining the Service's Endpoint Information

Overview

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

For more information on defining a service endpoint see Designing Artix Solutions.

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. Alt-click the **Services**.
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** 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**.

Building the Service

Overview

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

You will need to use a few Artix specific APIs to start the Artix bus and to access some of Artix's more advanced features. 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. Working in Java, is therefore, more standardized.

In this section

This section discusses the following topics:

Building a C++ Service	page 28
Building a Java Service	page 30

Building a C++ Service

Overview

Building a Web service using the Artix C++ APIs is a four step process. First you generate the skeleton code from the WSDL contract. Using the generated code as a base, you add the code for starting the Artix bus. Then you implement the service's business logic. Once the service is fully coded, you need to make the C++ application.

Generating code

To generate Artix C++ code for a Web service 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 `wsdltocpp` as shown below.

```
wsdltocpp -server -impl -sample -m NMAKE 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, a make file for the generated code and a sample `main()` for your service.

> **Note:** If you are on a UNIX system substitute `MAKE` for `NMAKE`. This will generate a UNIX make file.

For more information on the `wsdltocpp` tool see Developing Artix Applications with C++.

Writing the server main()

If you used the `-sample` flag when generating your code, the server's `main()` is mostly written for you. The code for the `main()` is generated into a file called *portTypeName*`ServerSample.cxx`. You will need to do the following to complete the `main()`:

1. Uncomment the `#include` statement for the server's impl. This line looks similar to the following.

```
\\#include "orderWidgetsImpl.h"
```

2. Uncomment the using statement for the types used by the server. This line looks similar to the following.

```
//using namespace COM_WIDGETVENDOR_WIDGETORDERFORM;
```

3. Uncomment the line of code that instantiates the impl object for your service. This line looks similar to the following.

```
// orderWidgetsImpl servant(bus);
```

4. Uncomment the lines of code the register the servant with the bus. They will look similar to the following.

```
// bus->register_servant(
//                         servant,
//                         "../widgets.wsdl",
//                         service_name
//                         );
```

5. Edit the second parameter of the `register_servant()` call to point to the location of the server's contract.

For more information on writing a C++ server `main()` see Developing Artix Applications with C++.

Implementing the service

The Artix 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 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 Artix functionality and working with the generated data types.

Building a Java Service

Overview

Building a Web service using the Artix Java APIs is a four step process. First you generate the skeleton code from the WSDL contract. Using the generated code as a base, you add the code for starting the Artix bus and registering your service's servant implementation. Then you implement the service's business logic. Once the service is fully coded, you need to build the Java application.

Generating code

To generate Artix Java code for a Web service 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 `wsdltojava` as shown below.

   ```
   wsdltojava -server -impl -ant 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, a simple main class for the service, 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 `javatowsdl` tool, see Developing Artix Applications in Java.

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

Starting the Artix bus

The Artix code generator creates a class the contains a `main()` for starting the Artix bus and registering a simple servant. This file is named based on the interface for which the code was generated. The code generator takes the value of the `name` attribute in the WSDL `<portType>` element that defines the interface and appends `Server` to it. For example, if the `<portType>` element defining the interface is named `orderWidgets`, the generated server main class will be named `OrderWidgetsServer.java`.

For simple cases, this class can be used without modification. However, most cases will require that you make some changes to this file so that type factories are registered, the server points to the proper WSDL contract, and the proper type of servant is registered with the bus.

You can make the following changes to the generated `main()` class:

1. If your service uses `anyType` elements, contexts, SOAP headers, or substitution groups uncomment the line that registers the type factories.

   The line of code for registering type factories will look like the following:

   ```
   bus.registerTypeFactory(new
    com.widgetvendor.widgetorderform.OrderWidgetsTypeFactory()
    );
   ```

2. Edit the WSDL path in the generated type factory to point to the proper location for your WSDL file.

   The line containing the WSDL path is similar to the following:

   ```
   private static String wsdlLocation = "../widgets.wsdl";
   ```

3. Edit the line for creating the servant to contain the proper path to the WSDL file. The WSDL path is the second parameter.

   The line containing the servant instantiation is similar to the following:

   ```
   Servant servant = new SingleInstanceServant(
       new com.widgetvendor.widgetorderform.OrderWidgetsImpl(),
       "file:../widgets.wsdl",
       bus);
   ```

4. If you want to use one of the other servant threading models, change the `SingleInstanceServant` to one of the alternative `Servant` types.

For more information see Developing Artix Applications with Java.

Implementing the service

The Artix 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 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 the 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 Artix functionality.

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

Building the service

You build the service 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

Overview

Once your service is built you need to configure the Artix runtime to launch the appropriate services to support the service. You also need to ensure that your service 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 services 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 Deploying and Managing 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

If you are not using logging, you simply launch the application as you would any other application.

Web Service Enabling Backend Services

Artix allows you to quickly expose backend services, that use a variety of middlewares, as Web 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 Web service. In most cases, there is no additional coding needed. The Artix bus does all of the work.

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

1. Creating a WSDL description of your service.
2. Defining a SOAP/HTTP endpoint for your service.
3. Configuring the Artix runtime to deploy the switching service.

In this chapter

This chapter discusses the following topics:

Describing a Service in WSDL	page 37
Defining a SOAP/HTTP Endpoint	page 49

Configuring and Deploying a Switching Service	page 52

Describing a Service in WSDL

Overview

The first step in web 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 XMLSchema. 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 WSDL documents see Designing Artix Solutions.

In this section

This section discusses the following topics:

Starting with CORBA IDL	page 38
Starting with a COBOL Copybook	page 40
Starting with a Java Class	page 43
Defining the Service's Endpoint Information	page 45

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 XMLSchema 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 45.

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

Using Artix desinger

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|Other** to open the **New** window.
3. Select **WSDL From IDL** from the **IONA Artix Designer** folder.
4. Click **Next>**.
5. Select the folder where you want to store the WSDL file.
6. Type the name of contract in the **File name** field.
7. Click **Next>**.
8. Enter the pathname for the IDL file.
9. 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.

10. 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 Designing Artix Solutions.

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 Designing Artix Solutions.

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 45.

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|Other**.
2. Select **WSDL From DataSet** from the **IONA Artix Designer** folder.
3. Click **Next>**.
4. Select the folder where you want to store the WSDL file.
5. Enter a name for the contract.
6. Click **Next>**.
7. Select **Fixed from a COBOL Copybook**.
8. Click **Next>**.
9. Enter the name for the generated `binding` element in the **Binding Name** field.
10. Enter the name for the generated `portType` element in the **Port Type Name** field.
11. Enter the namesapce for the generated contract's target namespace in the **Target Namespace** field.

12. Enter the namspace you wish to use as the target namespace for the generated contract's `schema` element in the **Schema Namespace** field.
13. 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.
14. Select the justification value to use in the generated fixed binding from the **Justification** drop-down list.
15. Enter the character encoding you wish to use for the generated fixed binding in the **Encoding** field.
16. Enter the character to use for padding on the wire in the **Padding** field.
17. Click **Next>**.
18. Click **Add** to add a new operation to the table.
19. Select the new operation from the table.
20. Click **Edit** to bring up the operation editing table.
21. Enter a name for the operation.
22. Select what type of operation you want to define from the **Style** drop-down list.
23. Enter a discriminator string for the operation in the **Discriminator** field.
24. Select one of the messages from tree on the left to bring up the message editing table.
25. Enter a name for the message.
26. Select the type of message from the **Type** drop-down list.
27. Enter the desired attributes for the message.
28. Click **Import COBOL Copybook** to bring up a file browser.
29. Select the copybook that defines the message from the file browser.
30. Repeat steps **24** through **29** for each message in the operation.
31. Repeat steps **18** through **30** for each operation in the service you are defining.
32. 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 Designing Artix Solutions.

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|Other** to open the **New** window.
2. Select **WSDL From Java** from the **IONA Artix Designer** folder.
3. Click **Next>**.
4. Select the folder where you want to store the WSDL file.
5. Enter a name for the contract.
6. Click **Next>**.
7. Click **Browse...** to open the Java class browser.
8. Enter a search string to locate the class you wish to import in the **Select a class to use** field.
9. Select the desired class from the list of types in the **Matching types:** list.
10. Click **OK**.
11. 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.
12. 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
```

For more information on the `javatowsdl` tool see Designing Artix Solutions.

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. 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 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 **tibrv** 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**.
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 Designing Artix Solutions.

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** 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 Designing Artix Solutions.

Configuring and Deploying a Switching Service

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 service 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/3.0/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 process 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 a switch 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 a switching process 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 3 shows an example of a configuration domain, `widgets.cfg`, that contains the information to launch a switching process.

Example 3: *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 3 you would use the following command.

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

Using Artix with .NET

Artix easily integrates with .NET applications.

Overview

Microsoft .NET is one of the more popular methods for developing web service. 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	page 56
Building an Artix Client for a .NET Service	page 59
Using Artix to Bridge from a Backend Service to .NET	page 61
Using Artix Services from .NET clients	page 63

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 client's written in .NET, the process is straight forward 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 61. If you want your clients to directly interact with services that use protocols other than SOAP/HTTP, you need to use Artix Connect.

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.

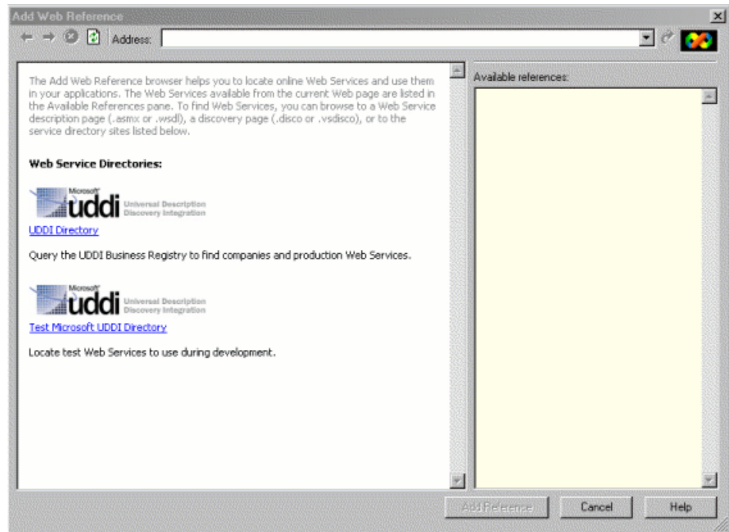

Figure 1: *Browsing For a Web Reference in Visual Studio*

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, you instantiate a proxy using the name of the service definition in the imported contract.

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

Example

Example 4 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 4: *.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 it's 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
   ```

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

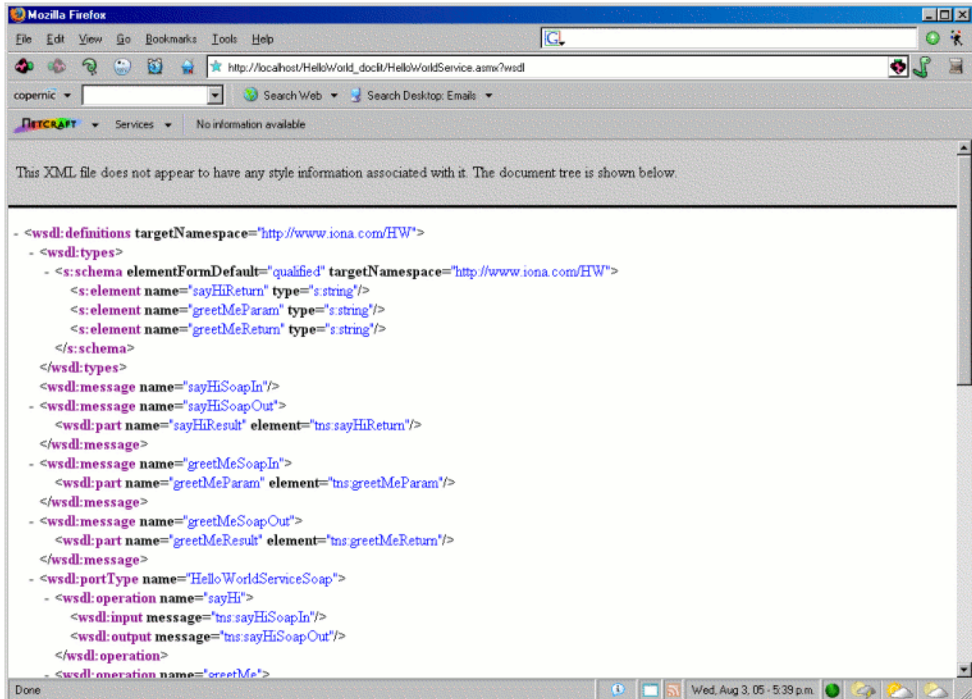

Figure 2: *.NET Service WSDL in a Browser*

2. Save the WSDL document to your local system using your browser's **File|Save As** option.
3. If you are using the 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 shown in Figure 3. This use case is very similar to the use case described in "Web Service Enabling Backend Services" on page 35. 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 56.

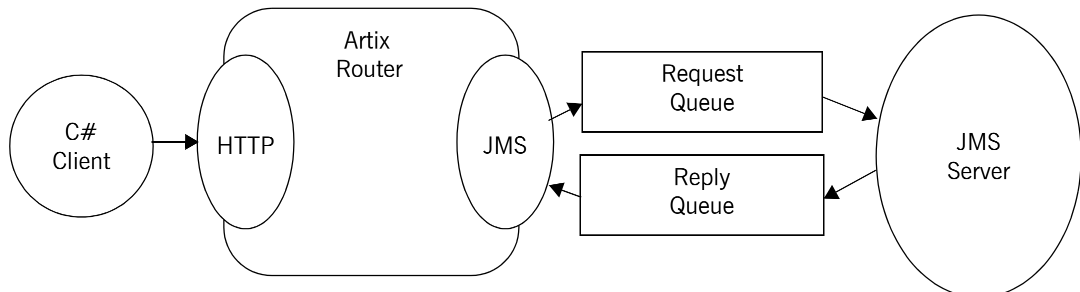

Figure 3: *Artix Bridge Between .NET and a JMS Service*

Procedure

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

1. Create an Artix contract describing you backend service as described in "Describing a Service in WSDL" on page 37.
2. Add a SOAP\HTTP endpoint definition to the service contract as described in "Defining a SOAP/HTTP Endpoint" on page 49.

3. Deploy an Artix switch as described in "Configuring and Deploying a Switching Service" on page 52.

   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, you instantiate a proxy using the name of the SOAP\HTTP endpoint definition in the imported contract.

When Visual Studio builds your client, it will automatically generate 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	page 64
Using the Artix Session Manager	page 68

Using the Artix Locator

Overview

The Artix locator is a light-weight 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 XMLSchema definition of an Artix reference, you can build a locator service proxy in .NET and write the logic to dig the appropriate information from the returned Artix reference.
- Use the `Bus.Services.dll` library provided with Artix to access the locator and decifer the returned Artix reference.

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

For more information on the Artix locator see Deploying and Managing 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 you new project and select **Add Reference** from the pop-up menu.

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

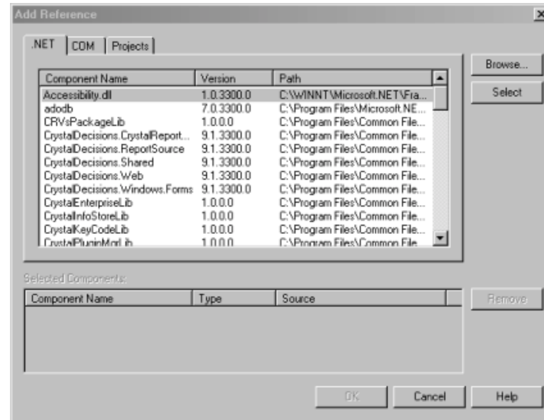

Figure 4: *Add Reference Window*

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 5.

Example 5: *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 6.

Example 6: *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 7.

Example 7: *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 8.

Example 8: *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 9 shows the code for a .NET client that looks up the HelloWorld service from a locator deployed at `localhost:8080`.

Example 9: *.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 light-weight 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 Deploying and Managing Artix Solutions.

What you need before starting

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

- The 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 a contract
- A locally accessible copy of a contract defining the service you desire to ultimately invoke upon.
- Microsoft's WSE 2.0 SP3

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 your new project and select **WSE 2.0 Properties**.

3. On the **General** tab of the properties dialog, check the **Enable this project for Web service enhancements** box.
4. Right-click the folder for your new project and select **Add Reference** from the pop-up menu.
5. Click **Browse** on **Add Reference** window.
6. In the file selection window browse to your Artix installation and select `utils\.NET\Bus.Services.dll`.
7. Click **OK** to return to the Visual Studio editing area.
8. Add an application configuration file to the project containing the XML shown in Example 10.

Example 10: *Application Configuration File*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <webServices>
      <soapExtensionTypes>
        <add type="Bus.Services.SessionIdExtension,Bus.Services"
             priority="1"
             group="0" />
      </soapExtensionTypes>
    </webServices>
  </system.web>
</configuration>
```

9. Right-click the folder for your new project and select **Add Web Reference** from the pop-up menu.
10. 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.
11. Click **Add Reference** to return to the Visual Studio editing area.
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 11.

Example 11: *Instantiating a Locator Proxy in .NET*

```
SessionManager sm = new SessionManager("http://localhost:8080");
```

The constructor's parameter is the HTTP address of a deployed session manager.

The `SessionManager` class has constructors that take the following:

- an Artix reference
- `ServiceDescription` containing the path to the session manager's contract.

15. Create a new Artix session by instantiating an instance of `Bus.Services.Session` as shown in Example 12.

Example 12: *Creating a new Session*

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

The constructor takes three parameters:

- An instantiated SessionManager object.
- A string identifying the group for which the client wants a session.
- 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 `SessionManager.get_all_endpoints()` as shown in Example 13.

Example 13: *Getting the References for the Managed Endpoints*

```
Reference[] ref = sm.get_all_endpoints(s.GetSessionId());
```

    `get_all_endpoints()` 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 using the contructor with `Wse` apended to it as shown in Example 14.

Example 14: *Creating a WSE Enabled Proxy.*

```
SOAPServiceWse pxy = new SOAPServiceWse();
```

18. Change the value of the proxy's `.Url` member to the SOAP address of one of the Artix reference returned from the session manager as shown in Example 8.

Example 15: *Changing the URL of a .NET Service Proxy to Use a Reference*

```
pxy.Url = ReferenceHelper.GetSoapAddress(ref[0]);
```

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

19. Create a `SessionFilter` object as shown in Example 16.

Example 16: *Creating a SessionFilter*

```
Bus.Services.SessionFilter sFilter = new
    Bus.Services.SessionFilter(s);
```

20. Instruct the proxy to include the session header in all of its requests by adding a `SessionFilter` to the proxy's output pipeline as shown in Example 17.

Example 17: *Setting a Proxy's Session Header*

```
pxy.Pipeline.OutputFilters.Add(sFilter);
```

    Once you have called this method on the proxy 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.

21. Make requests on the service as you would normally.
22. When you are done with the service, end the session by calling `endSession()` on the `Session` object.

Example

Example 18 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 18: *.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 = ReferenceHelper.GetSoapAddress(r[0]);

     proxy.SetSession(s);

      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();
    }
  }
}
```

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	page 77
Client and Server on Different MQ Servers	page 78
Client on MQ Client and Server on MQ Server	page 81
Using a Remote MQ Server	page 83
Using a Remote MQ Server from Full MQ Installations	page 85

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 19, the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

Example 19: *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 20, the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

Example 20: *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 21 the `<mq:client>` and the `<mq:server>` elements include the attributes that configure the transport.

Example 21: *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 81 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 22 the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

Example 22: *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 83 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 23 the `<mq:client>` and the `<mq:server>` elements include attributes that configure the transport.

Example 23: *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 6

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	page 89
Transforming a Sequence into a String	page 91
Modifying a Simple Sequence	page 93
Working with Nested Input Sequences	page 96
Working with Attributes in an Input Message	page 99
Working with Attributes in the Output Message	page 102
Working with Nested Sequences in an Output Message	page 105
Using Multiple Templates in a Script	page 108

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 portType `transformer`, the operation `transformer_operation` 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 portType `transformer`, the operation `transformer_operation` 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 24 shows the template for the XSLT scripts used later in the chapter.

Example 24: *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>
  </xsl: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 25.

Example 25: *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="client_request_type"
               type="tns:complex_request_type"/>
      <element name="transformer_reply_type" 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:client_request_type"
          name="client_request"/>
  </message>
```

Example 25: *Contract Fragment for Sequence to String*

```
  <message name="transformer_reply_message">
    <part element="tns:transformer_reply_type"
          name="transformer_reply"/>
  </message>
  <portType name="transformer">
    <operation name="transformer_operation">
      <input message="tns:client_request_message"
             name="transformer_operationRequest"/>
      <output message="tns:transformer_reply_message"
              name="transformer_operationResponse"/>
    </operation>
  </portType>
```

The XSLT script

Example 26 shows an XSLT script that simply concatenates the first and last name values from the incoming data into the output data string.

Example 26: *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 27.

Example 27: *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="client_request_type"
             type="tns:complex_request_type"/>
    <element name="transformer_reply_type"
             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:client_request_type"
          name="client_request"/>
  </message>
```

Example 27: *Contract Fragment for Modifying a Sequence*

```
  <message name="transformer_reply_message">
    <part element="tns:transformer_reply_type"
          name="transformer_reply"/>
  </message>
  <portType name="transformer">
    <operation name="transformer_operation">
      <input message="tns:client_request_message"
             name="transformer_operationRequest"/>
      <output message="tns:transformer_reply_message"
              name="transformer_operationResponse"/>
    </operation>
  </portType>
```

The script

The XSLT script shown in Example 28 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 28: *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 29.

Example 29: *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="client_request_type"
             type="tns:complex_request_type"/>
    <element name="transformer_reply_type"
             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 29: *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:client_request_type"
          name="client_request"/>
  </message>
  <message name="transformer_reply_message">
    <part element="tns:transformer_reply_type"
          name="transformer_reply"/>
  </message>
  <portType name="transformer">
    <operation name="transformer_operation">
      <input message="tns:client_request_message"
             name="transformer_operationRequest"/>
      <output message="tns:transformer_reply_message"
              name="transformer_operationResponse"/>
    </operation>
  </portType>
```

The XSLT script

The XSLT script for this example, shown in Example 30, is very similar to the script in Example 28. 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 30: *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 30: *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 31.

Example 31: *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="client_request_type"
             type="tns:complex_request_type"/>
    <element name="transformer_reply_type"
             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 31: *Contract Fragment for Input 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="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:client_request_type"
        name="client_request"/>
</message>
<message name="transformer_reply_message">
  <part element="tns:transformer_reply_type"
        name="transformer_reply"/>
</message>
<portType name="transformer">
  <operation name="transformer_operation">
    <input message="tns:client_request_message"
           name="transformer_operationRequest"/>
    <output message="tns:transformer_reply_message"
            name="transformer_operationResponse"/>
  </operation>
</portType>
```

The XSLT Script

While the syntax for accessing the members of the nested sequence in the XSLT script shown Example 32 looks similar to the syntax used in Example 30, notice the `@` used when referencing the `sex` attribute. The `@` specifies that the value to be matched in an attribute, not an element.

Example 32: *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="/">
    <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 33.

Example 33: *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="client_request_type"
             type="tns:complex_request_type"/>
    <element name="transformer_reply_type"
             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 33: *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:client_request_type"
        name="client_request"/>
</message>
<message name="transformer_reply_message">
  <part element="tns:transformer_reply_type"
        name="transformer_reply"/>
</message>
<portType name="transformer">
  <operation name="transformer_operation">
    <input message="tns:client_request_message"
           name="transformer_operationRequest"/>
    <output message="tns:transformer_reply_message"
            name="transformer_operationResponse"/>
  </operation>
</portType>
```

The XSLT Script

Example 34 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 34: *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 35.

Example 35: *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="client_request_type"
             type="tns:complex_request_type"/>
    <element name="transformer_reply_type"
             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 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:client_request_type"
        name="client_request"/>
</message>
<message name="transformer_reply_message">
  <part element="tns:transformer_reply_type"
        name="transformer_reply"/>
</message>
<portType name="transformer">
  <operation name="transformer_operation">
    <input message="tns:client_request_message"
           name="transformer_operationRequest"/>
    <output message="tns:transformer_reply_message"
            name="transformer_operationResponse"/>
  </operation>
</portType>
```

The XSLT script

Unlike in the previous examples, the XSLT script required to create the output message, shown in Example 36, must now include tags to create the `first_name` and `last_name` elements derived from the nested sequence.

Example 36: *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">
```

Example 36: *Creating a Nested Output Sequence*

```
  <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>
      <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 37 shows one way of separating the directives used in Example 36.

Example 37: *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 37: *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 7

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

Example 38: *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 39 shows the code for implementing such a `Handler`.

Example 39: *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 39: *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 39 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 40 shows the client code for setting the username and password on the `Stub`.

Example 40: *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 8

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	page 118
When Only One Side Uses Unmapped XML Data	page 121

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 41 illustrates how the `xsd:any` type can be used within a message part.

Example 41: *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>
<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 42.

Example 42: *Generated Class Using xsd:any*

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

public class UnmappedType
{
    private SOAPElement _any;

    public SOAPElement get_any() {
        return _any;
    }
```

Example 42: *Generated Class Using xsd: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 43, this class will also create and populate a `SOAPElement` instance with the content obtained from an XML file.

Example 43: *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 44.

Example 44: *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 45 shows a contract for a client that uses unmapped XML data.

Example 45: *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 45: *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 43. 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 46.

Example 46: *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 47.

Example 47: *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 47: *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 48, its contents should match the contents of the original `employee.xml` file.

Example 48: *Getting the Employee Object*

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

Glossary

A

anyType

anyType is the root type for all XML Schema types. All of the primitive types are derivatives of this type, as are all user defined complex types.

Artix message context

An Artix message context is a special message context that is used by Artix to store and transmit transport details and message header information. They contain two context containers. One for storing data about requests and one for storing data about replys.

Artix reference

An Artix reference is a Java object that fully describes a running Artix service. References can be passed between Artix endpoints as operation parameters and are used extensively by the Artix locator.

B

Binding

A binding maps an operation's messages to a payload format. Bindings are defined using the WSDL `<binding>` element.

Bus

See Service Bus.

C

Choice complex type

A choice complex type is an XMLSchema construct defined by using a `<choice>` element to constrain the possible elements in a complex type. When using a choice complex type only one of the elements defined in the complex type can be valid at a time.

Classloader firewall

The classloader firewall provides a user configurable way to block the Artix Java runtime from classes on a system's classpath.

Contract

An Artix contract is a WSDL file that defines the interface and all connection information for that interface.

A contract contains two components: *logical* and *physical*. The logical component defines things that are independent of the underlying transport and wire format such as abstract definitions of the data used and the interface.

The physical component defines the wire format, middleware transport, and service groupings, as well as the mapping between the operations defined in the interface and the wire formats, and the buffer layout for fixed formats and extensors.

D

Discriminator

A discriminator is a data element created to support the mapping of a choice complex type to a Java object. The discriminator element identifies the valid element in a choice complex type. See also Choice complex type.

Dynamic proxy

A dynamic proxy is a Java construct introduced in version 1.3 by Sun Micosystems. As specified by the JAX-RPC specification, Artix uses a dynamic proxy to connect to remote services. For more information, go to http://java.sun.com/reference/docs/index.html.

E

Embedded deployment

An embedded deployment is a deployment mode in which an application creates an endpoint, either by invoking Artix APIs directly, or by compiling and linking Artix-generated stubs and skeletons to connect client and server to the service bus.

Endpoint

The runtime incarnation of a service defined in an Artix contract. When using the Artix Java APIs, an endpoint is activated when you register a servant with the Artix bus. See also Service.

F

Facet

A facet is a rule used in the derivation of user defined simple types. Common facets include `length`, `pattern`, `totalDigits`, and `fractionDigits`.

Factory pattern

The factory pattern is a usage pattern where one service creates and manages instances of another service. Typically, the factory service returns references to the services it creates.

Fault message

A fault message is the WSDL construct used to define error messages, or exceptions, passed between a service and its clients. They are defined using a `<fault>` element in a WSDL contract.

H

Handler

`Hanlder` is the Java interface that a developer must implement to create a message handler. It has has methods for processing both request and response messages. Artix provides a `GenericHandler` class to provide a template for implemeting message handlers.

I

Input message

An input message is the WSDL construct for defining the messages that are sent from a client to a service and are specified using an `<input>` element in a WSDL contract. When mapped into Java, the parts of the input message are mapped into a method's parameter list.

Interface

An interface defines the operations offered by a service. Interfaces are defined in an Artix contract using the WSDL `<portType>` element. When mapped to Java, an interface results in the generation of an object with methods for each of the operations defined in the interface. See also Operation.

J

JAX-RPC

JAX-RPC (Java API for XML-Based RPC) is the Java specification upon which Artix based it Java API and data type mappings. For more information go to http://java.sun.com/xml/jaxrpc/overview.html.

L

List type

A list type is a data type defined as consisting of a space separated list of primitive type elements. For example, "1 2 3 4 5" is a valid value for a list type. They are defined using a `<xsd:list>` element.

Logical contract

The logical contract defines components that are independent of the underlying transport and wire format. These include the type definitions and the interface definitions. WSDL elements found in the logical contract include: `<portType>`, `<operation>`, `<message>`, `<type>`, and `<import>`.

M

Message

In Artix, a message is any data passed between two endpoints. Messages are defined in an Artix contract using the WSDL `<message>` element and are used for the input, output, and fault messages that define an operation. After a message has been associated with an operation, it can be bound to any payload format supported by Artix. See also Fault message, Input message, and Output message.

Message-level handler

A message-level handler is a message handler that processes messages between the Artix binding to the Artix transport.

Message context

A message context is a bus container used by applications to store metadata properties. These properties store information about the message being sent went an operation is invoked. Artix uses the message context to store headers and transport information. See also Artix message context.

Message handler

A message handler is a Java class responsible for intercepting a message along the message chain and performing some processing on the raw message data. See also Handler.

O

Operation

An operation defines a specific interaction between a service and a client. It is defined in an Artix contract using the WSDL `<operation>` element. Its definition must include at least one input or output message. When mapped into Java, an operation generates a method on the object representing the interface in which it is defined.

Output message

An output message is the WSDL construct for defining the messages that are sent from a service to a client and are specified using an `<output>` element in a WSDL contract. When mapped into Java, the parts of the output message are mapped as described in the JAX-RPC specification.

P

Payload format

A payload format is how data is packaged to be sent on the wire. Examples of payload formats supported by Artix include SOAP, TibMsg, and fixed record length data. Data is bound to a payload format in an Artix contract using the WSDL `<binding>` element.

Physical contract

The physical contract defines the bindings and transport details used by the endpoints defined by an Artix contract. WSDL elements found in the physical contract include: `<binding>`, `<service>`, and `<port>`.

Plug-in

A plug-in is a module that Artix loads at runtime to provide a set of features. All of the bindings and transports supported by Artix are implemented as plug-ins. In addition, message handlers are implemented as plug-ins.

R

Reply

A reply is the message returned by a service to a client in response to a request from the client. See also Output message.

Request

A request is a message sent from a client to a service asking for the service to do work. See also Input message.

Request-level handler

A request-level handler is a message handler that processes messages between the Artix binding and the user's application code.

Response

See Reply.

S

Servant

A servant is a Java object that wraps the implementation object generated from an interface. The servant wrapper enables the bus to associate the implementation object with the physical details specified in its contract's service definition and to manage the object.

Service

A service is the contract definition of an Artix endpoint. It combines the logical definition of an interface, the binding of the interface's operations to a payload format, and the transport details used to expose the interface. A service is defined using a WSDL `<port>` element.

Service Bus

The infrastructure that allows service providers and service consumers to interact in a distributed environment. Handles the delivery of messages between different middleware systems. Also known as an Enterprise Service Bus.

Service proxy

A service proxy is a proxy created by an Artix client to connect to a remote service. See also Dynamic proxy.

Service template

A service template is a WSDL service definition that serves as the model for the clones created for a transient reference. They must fully define all of the details, except the address, of the transport used by the transient servant. The address provided in the service template must be a wildcard value.

Standalone deployment

Standalone deployment is a deployment mode in which an Artix instance runs independently of the endpoints it is integrating.

Static servant

A static servant is a servant whose physical details are linked to a `<port>` definition in the contract associated with the application.

Stub interface

Artix service proxies implement the `javax.xml.rpc.Stub` interface. The Stub interface provides access to a number of low-level properties used to connect the proxy to a remote service. These properties can be used to get the Artix bus from client applications and set HTTP connection properties.

T

Transient servant

A transient servant is a servant whose physical details are cloned from a `<port>` definition in the contract associated with the application.

Transport

A transport is the network protocol, such as HTTP or IIOP, that is used by an endpoint. The transport details for an endpoint are defined inside of the WSDL `<port>` element defining the endpoint.

Type factory

A type factory is a Java class generated to support the use of XMLSchema `anyTypes` and SOAP headers in Java.

W

WSDL

WSDL (Web Services Description Language) is an XML format for describing network services as a set of endpoints. Artix uses WSDL as the syntax for its contracts.

In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data binding formats. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- Types -- a container for data type definitions using some type system (such as XMLSchema).
- Message -- an abstract, typed definition of the data being communicated.

- Operation -- an abstract definition of an action supported by the service.
- Port Type -- an abstract set of operations supported by one or more endpoints.
- Binding -- a concrete protocol and data format specification for a particular port type.
- Port -- a single endpoint defined as a combination of a binding and a network address.
- Service -- a collection of related endpoints.

For more information go to http://www.w3.org/TR/wsdl.

WSDL <binding>

See Binding and Payload format.

WSDL <fault>

See Fault message.

WSDL <message>

See Message.

WSDL <operation>

See Operation.

WSDL <port>

See Service.

WSDL <portType>

See Interface.

WSDL <service>

A WSDL `<service>` element is a collection of WSDL `<port>` elements.

X

XML Schema

XML Schema is a language specification by the W3C that defines an XML meta-language for defining the contents and structure of XML documents. It is used as the native type system for Artix. For more information go to http://www.w3.org/XML/Schema.

Index

C

C++
  generating from the designer 5
  generating with wsdltocpp 6
C++ client
  adding business logic 8
  generating code 6
  intantiating a proxy 7
  writing a main() 6
C++ server
  generating code 28
  implementing the servant 29
  writing main() 28
client
  adding business logic 8, 13
  configuration 14
  generating C++ code 6
  generating Java code 10
  instantiating a Java proxy 11
  intantiating a C++ proxy 7
  required Java classes 13
  steps for building 1
  writing a C++ main() 6
  writing a Java main() 11
coboltowsdl 42
com.iona.webservices.saaj.util.SAAJUtils 120
complexType
  adding with the designer 19
configuration
  contract locations 53
  creating 52
  plug-ins 53
container
  deploying 53
copybook
  converting to WSDL 42
  importing 40
CORBA
  defining an endpoint 45

D

designer
  adding a CORBA endpoint 45
  adding a JMS endpoint 47
  adding a message 22
  adding an HTTP endpoint 50
  adding an HTTP port 26
  adding an interface 23
  adding an MQ endpoint 45
  adding an operation 24
  adding a portType 23
  adding a route 50
  adding a SOAP binding 25, 49
  adding a Tibco endpoint 47
  adding a Tuxedo endpoint 46
  adding unsupported types 21
  creating a project 2, 17
  defining a complexType 19
  defining an element 20
  defining a simpleType 18
  importing a copybook 40
  importing IDL 38
  importing Java 43
  starting 2, 17
  using source view 21

E

element
  adding with the designer 20
endpoint
  CORBA 45
  defining 45
  HTTP 50
  JMS 47
  MQ 45
  Tibco 47
  Tuxedo 46

H

HTTP
  adding an endpoint 50

I

IDL
  converting to WSDL 39

idltowsdl 38
importing IDL
 idltowsdl 39
 using the designer 38
interface
 adding an operation 24
 adding with the designer 23
IOR
 from a file 39
 from a naming service 38

J
Java
 converting to WSDL 43
 generating from the designer 9
 generating with wsdltojava 10
Java client
 adding business logic 13
 generating code 10
 initializing the bus 11
 instantiating a proxy 11
 registering type factories 11
 required classes 13
 writing a main() 11
Java server
 generating code 30
 implementing a servant 32
 registering type factories 31
 required classes 32
 writing the main() 31
javax.xml.soap.SOAPElement 118
JMS
 adding an endpoint 47

M
message
 adding with the designer 22
MQ
 adding an endpoint 45
 deployment scenarios 75

O
operation
 adding with the designer 24
org.w3c.dom.Document 118
org.w3c.dom.Node 118

P
payload format
 SOAP 25, 49, 50
portType
 adding an operation 24
 adding with the designer 23

R
routing
 adding a route 50

S
server
 configuring 33
 creating a project 17
 generating C++ 28
 generating Java code 30
 implementing a C++ servant 29
 implementing a Java servant 32
 registering Java type factories 31
 required Java classes 32
 steps for building 15
 writing a C++ main() 28
 writing a Java main() 31
service enabling 35
simpleType
 adding with the designer 18
SOAP
 adding a binding 49, 50

T
Tibco
 adding an endpoint 47
transports
 HTTP
  adding 26
Tuxedo
 adding an endpoint 46

W
wsdltocpp 6, 28
wsdltojava 10, 30, 43
wsdltosoap 50

X
XML
 Artix code 120

using 118
XMLSchema
complexType
adding 19
constructs without wizards 21
editing 21
element
adding 20
simpleType
adding 18