IONA®

Artix™

Getting Started with Artix Java

Version 1.2, October 2003

Making Software Work Together™

Updated: 07-Nov-2003

Contents

List of Figures

Preface

Overview

Getting Started with Artix Java is for use by anyone who needs to understand the concepts and terms used in the IONA Artix product, but more specifically with Artix Java, the Java flavor of Artix Encompass.

Artix Java is provided for evaluation use only; it is not a fully supported product feature. For this reason, all Artix Java-related files are stored in the *contrib* directory of the Artix product kit.

Audience

This manual is geared for first time Artix users. It is assumed that the reader is familiar with the middleware systems discussed in this manual.

Organization of this guide

This guide is divided as follows:

- “Artix Java Concepts” provides general information about Artix Java and how it is used.
- "Artix Java in Action: a demo" presents a walk through of how to solve an integration problem with Artix Java, using a mixture of the command line tools and the Artix Designer.

Related documentation

The document set for IONA Artix includes the following:

- *Getting Started with Artix Encompass*
- *Getting Started with Artix Relay*
- *Artix User’s Guide*
- *Artix Installation Guide*
- *Artix Tutorial*

- *Artix C++ Programming Guide*
- *Artix Security Guide*
- *Artix Thread Library Reference*

The latest updates to the Artix documentation can be found at http://iona.com/support/docs/artix/1.2/index.xml.

Online help

The Artix Designer includes comprehensive online help, providing:

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

There are two ways to access the online help: via the Help menu in the Artix Designer, or by clicking the Help button on any interface dialog.

Additional resources

The IONA Knowledge Base contains helpful articles, written by IONA experts, about Orbix and other products. You can access the knowledge base at the following location: (http://www.iona.com/support/knowledge_base/index.xml)

The IONA Update Center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`

Italic	Italic words in normal text represent *emphasis* and *new terms*.
	Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:
	`% cd /users/`*`your_name`*
	Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the DOS 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.
`|`	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Artix Java Concepts

Artix Java enables your organization to realize all the benefits of a C++ runtime environment, without having to write any C++ code.

In this chapter

This chapter discusses the following topics:

Introduction to Artix

Overview

Artix is a new approach to application integration, one that exploits the middleware technologies and products already present within an enterprise. Artix provides a rapid integration approach that increases operational efficiencies and makes it easier for an enterprise to adopt or extend a Service Oriented Architecture (SOA).

The Artix Product family is a suite of Enterprise Middleware Integration (EMI) products built on a proven infrastructure and designed to help lower operating costs and improve corporate efficiency. The products are:

- Artix Encompass - an enterprise Web Service product that extends enterprise qualities of service (transactions, security, routing, high availability, and management, for example) to Web Services applications and enables the rapid creation and deployment of EMI solutions using Web Services technology.
- Artix Relay - a middleware interoperability product, whch enables the seamless interoperability of diverse middleware platforms without the use of messaging hubs or intermediate formats, and without changing those systems.

Artix Java **- an innovation**

Everyone knows that Web Services are THE big thing in application integration. Not surprisingly, this shift towards Web Services has resulted in companies being well endowed with Java programmers. Until now this has been a potential problem when trying to address enterprise integration, in that the majority of enterprise applications are written in C++ - an uncharted world for the majority of Java developers.

Enter Artix Java, providing all the benefits of a C++ runtime environment without having to retrain your current Java developers, or worse still, supplement them with a team of C++ developers. Even better, Artix Java enables painless integration with C++ APIs for a variety of message transports.

Put simply, Artix Java provides a Java "wrapper" around the Artix C++ libraries. It enables your developers to use Java to create Artix-based applications and gain the full advantages of Artix C++ runtime features.

Supported transports/protocols

A transport is an on-the-wire format for messages. A protocol is a transport that is defined by an open specification. Thus MQ and Tuxedo are transports, while HTTP and IIOP are protocols. Throughout the Artix interface and documentation, both protocols and transports are referred to as transports. Artix supports the following message transports:

- HTTP
- BEA Tuxedo
- IBM WebSphere MQ (formerly MQSeries)
- TIBCO Rendezvous™
- IIOP
- IIOP Tunnel

An in-depth discussion of the differences between IIOP and IIOP Tunnel is beyond the scope of this chapter. Basically, IIOP Tunnel lets you use IONA's Application Server Platform infrastructure (if present in your enterprise) and exploit its qualities of service in support of non-CORBA payloads. Usage of all these transports is described fully in the *Artix User's Guide*.

Supported payload formats

Artix can automatically transform between the following payload formats:

- G2++
- FML – Tuxedo format
- CORBA (GIOP) – CORBA format
- FRL – fixed record length
- VRL – variable record length
- SOAP
- TibrvMsg - TIBCO/Rendezvous format

The mapping of logical data items between payload formats is supported by Artix tools.

Benefits of Artix

The Artix approach differs from the approach used by Enterprise Application Integration (EAI) products. The EAI approach typically uses a "canonical" format in an EAI hub. All messages are transformed from a source application's native format to this canonical format, and then transformed again to the format of the target application. Each application requires two adapters that translate to and from the canonical format.

However, requiring two translations for every message incurs high overhead. Many enterprises prefer high-performance solutions that directly transform a small set of message types over a more general solution with lower performance.

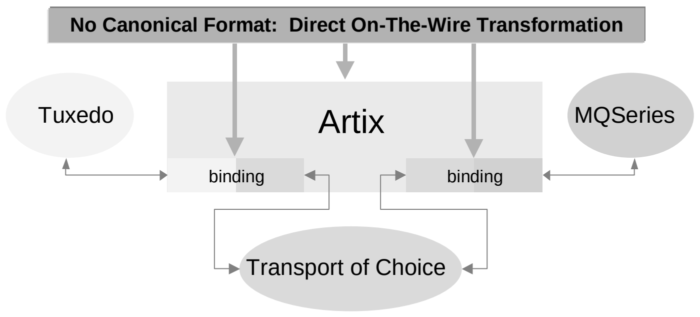


Figure 1: *Artix High-Performance Architecture*

Because Artix connects applications at the middleware transport level, Artix connections resemble the way network switches connect telephones. Like network switching, Artix hides the details of the connection and provides very high performance.

Artix Contracts

Overview

The Web Services Definition Language (WSDL) is used to describe the characteristics of the Service Access Points (SAPs) of an Artix connection. The SAP is the mechanism, and the points at which individual service providers and consumers connect to the service bus.

By defining characteristics like service operations and messages in an abstract way — independent of the actual transport or protocol used to implement the SAP — these characteristics can be bound to a variety of a specific protocols and formats. In fact, Artix allows an abstract definition to be bound to multiple specific protocols and formats. This means that the same definitions can be reused in multiple implementations of a service.

Artix contracts define the services exposed by a set of systems, the payload formats and transports available to each system, and the rules governing how the systems interact with each other. The most simple Artix contract defines a set of systems with a shared interface, payload format, and transport. Artix contracts, however, can define very complex integration scenarios.

WSDL concepts

Understanding Artix contracts requires some familiarity with WSDL, including the definitions of the following terms:

A **WSDL type** provides data type definitions used to describe messages.

A **WSDL message** is an abstract definition of the data being communicated and each part of a message is associated with defined types.

A **WSDL operation** is an abstract definition of the capabilities supported by a service, and is defined in terms of input and output messages.

A **WSDL Port Type** is a set of abstract operation descriptions.

A **WSDL binding** associates a specific protocol and data format for operations defined in a port type.

A **WSDL Port** specifies a network address for a binding, and defines a single communication endpoint.

A **WSDL service** specifies a set of related ports.

The Artix contract

An Artix contract is specified in WSDL and conceptually divided into logical and physical components. The logical contract specifies things that are independent of the underlying transport and wire format; it fully specifies the data structure and the possible operations or interactions with the interface. The Logical contract allows Artix to generate skeletons and stubs without having to define the physical characteristics of the connection (wire format and transport).

The physical component of an Artix contract defines:

- The wire format, middleware transport, and service groupings
- The connection between the PortType 'operations' and wire formats
- Buffer layout for fixed formats
- Artix extensions to WSDL

Example 1: *Artix WSDL Contract Elements*

```
Logical Contract:

<Schema>
<Type>                 (analogous to typedefs)
<Message>              (analogous to parameters)
<PortType>             (analogous to class or CORBA interface definition)
 <Operations>          (analogous to methods)

Physical Contract:

<Binding>              (payload format)
<Services>             (groups of ports)
  <Port>               (transport)
<Route>                (rules governing system interaction)
```

Payload Formats

A payload format controls the layout of a message delivered over a transport. The WSDL definition of a Port and its binding together associate a payload format with a transport. A binding can be specified in the logical portion of an Artix contract (`portType`), which allows for a logical contract to have multiple bindings and thus allow multiple on-the-wire formats to use the same contract.

The Artix Designer

Overview

The Artix Designer is a tool for creating and managing Artix contracts. It provides editors for creating contracts from standard WSDL files as well as from CORBA IDL files. The Designer also makes it easy to define new data types, logical interfaces, payload bindings, and transports by providing wizards to walk you through each step.

The Artix Designer generates all of the Artix components you need to complete your project, including:

- Artix contracts describing each of the services in your system.
- An Artix contract describing how Artix integrates your services.
- Any Artix stub or skeleton code needed to write Artix application code.
- The necessary configuration information to deploy your Artix instances.

In addition, the Artix Designer can also generate CORBA IDL from any contracts that have a CORBA binding.

Artix Welcome dialog

When you start the Artix Designer, your first interaction is with the Welcome dialog, as shown in Figure 2, where you can specify whether to create a new project, open an existing project, or go straight to the Designer.

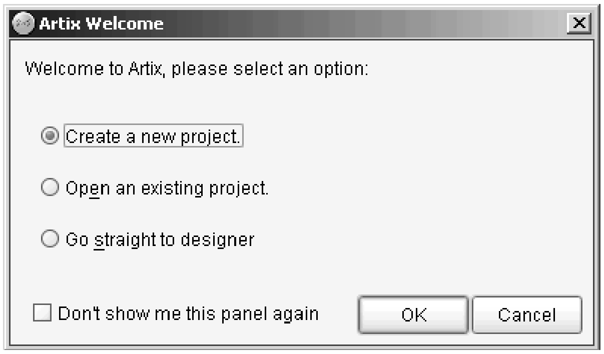

Artix Welcome
Welcome to Artix, please select an option:
Create a new project.
Open an existing project.
Go straight to designer
Don't show me this panel again
OK
Cancel

Figure 2: *Welcome Dialog*

Project Tree

On the left side of the Designer is the Project Tree. The Project Tree lists all of the System Diagram components with nodes for generating code, generating deployment information, and, if you are using CORBA, generating IDL. The Project Tree, as shown in Figure 3, also lists all of the contracts imported into your project.

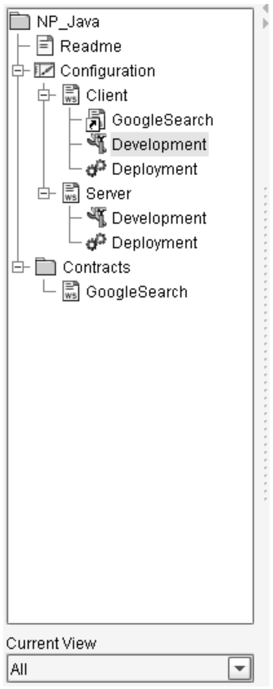


Figure 3: *Project Tree*

The drop down list at the bottom of the tree filters the amount of detail shown in the tree at a time. The default is to show all information about the project. You can select to view only the contracts imported into the project or just the system components.

System Diagram

The first layer of information you receive from the Artix Designer is a graphical representation of your project's configuration. This view is invoked by selecting **Configuration** in the Project Tree, and is called the System Diagram - see Figure 4 for an example.

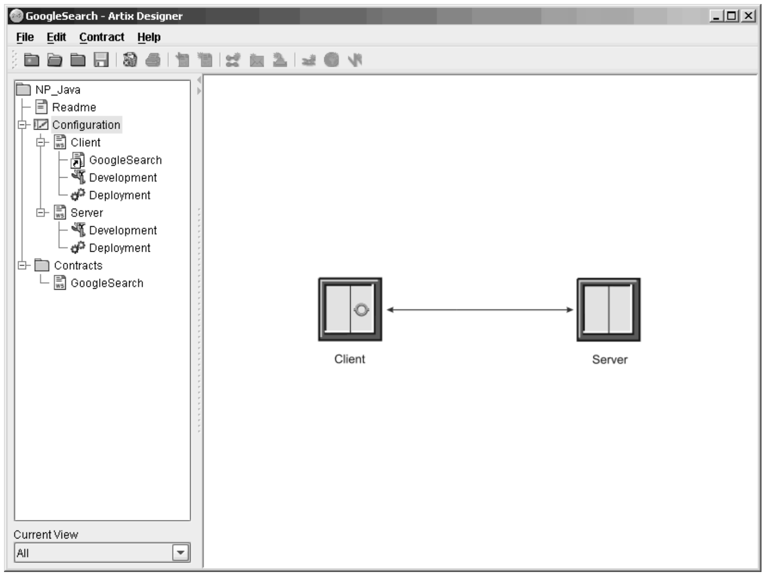


Figure 4: *The System Diagram*

This example of a System Diagram shows an Encompass configuration of a client and a server, with Artix embedded in the client entity.

An example of an *Artix Relay* configuration would contain a separate Artix entity in between the two system elements.

You cannot currently perform any actions in the System Diagram view.

Contract Editor

The Contract Editor (Figure 5) is the real engine room of the Artix Graphical User Interface (GUI). It serves two purposes - firstly it provides a way for you to navigate around the various components of your WSDL contract. Secondly, it provides you with access to editing tools to add, or change, Artix contract components.

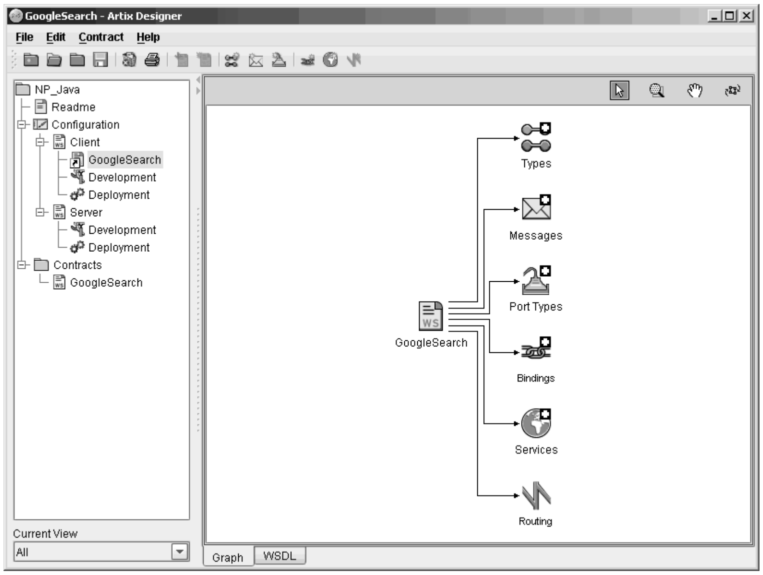


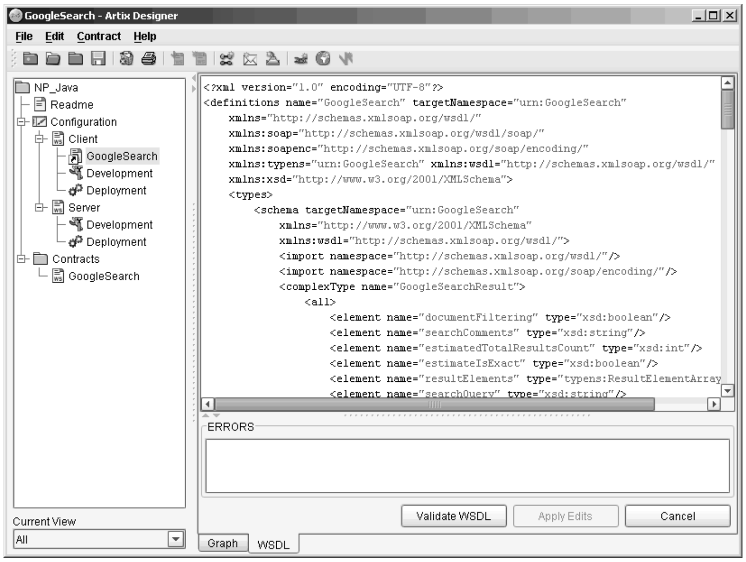
Figure 5: *The Contract Editor*

You'll notice that the icons representing the contract elements (types, messages, services, etc) in this diagram sometimes have a small plus sign attached — this indicates that the element has "children" — you can expose those children - or the actual types, messages, etc — for the contract by double clicking on the element icon. You can then view or edit the individual items directly from the Contract Editor.

Working with the WSDL

The Contract Editor also provides the option for you to view and edit the contract WSDL directly instead of working through the graphical representation as previously described.

To access the WSDL view of the contract, as shown in Figure 6, click on the WSDL tab at the bottom of the Contract Editor panel.



```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="GoogleSearch" targetNamespace="urn:GoogleSearch"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:typens="urn:GoogleSearch" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types>
        <schema targetNamespace="urn:GoogleSearch"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <import namespace="http://schemas.xmlsoap.org/wsdl/"/>
            <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
            <complexType name="GoogleSearchResult">
                <all>
                    <element name="documentFiltering" type="xsd:boolean"/>
                    <element name="searchComments" type="xsd:string"/>
                    <element name="estimatedTotalResultsCount" type="xsd:int"/>
                    <element name="estimateIsExact" type="xsd:boolean"/>
                    <element name="resultElements" type="typens:ResultElementArray
                    <element name="searchQuery" type="xsd:string"/>
```

Figure 6: *The Contract Editor - WSDL View*

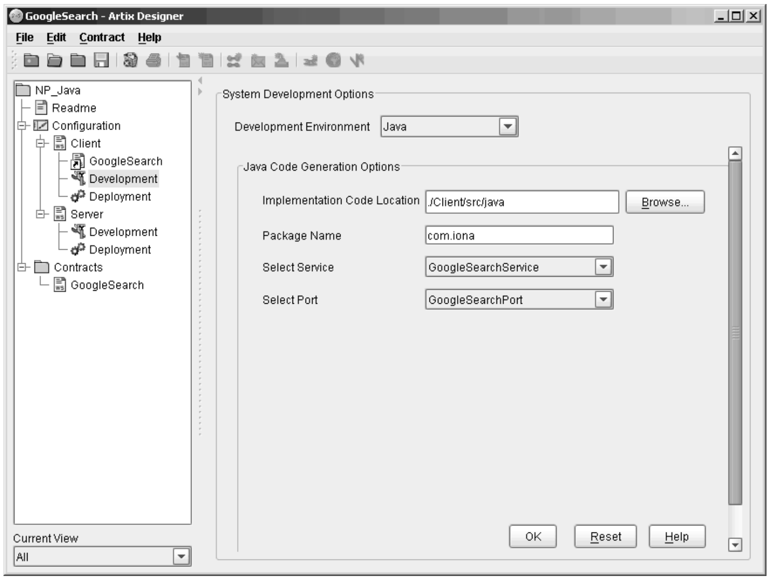
Working in the WSDL view of the contract requires a sound knowledge of WSDL — be aware that if you change the WSDL it could easily invalidate your contract.

If you make a change to the WSDL that does cause a problem, errors are identified in a separate error panel directly under the WSDL so that you can easily identify the exact position of the problem within the WSDL file.

Development Tool

The Development Tool is invoked by selecting the **Development** icon under one of the services in the project tree. Using this tool, shown in Figure 7, you can generate Artix stub and skeleton code for the interfaces defined by the selected service's contract. The tool will also generate a makefile and sample server and client mainlines for you.

The code generation options available to you on this panel are Java, CORBA, or C++.

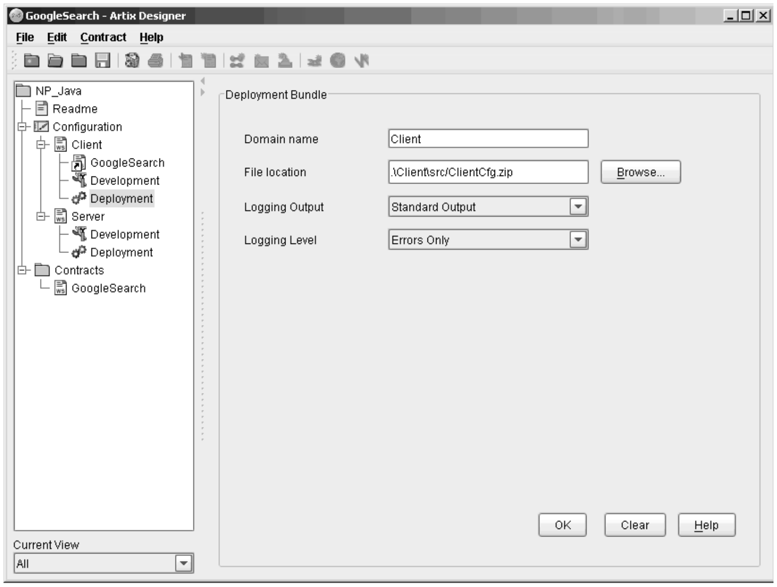


Development Environment: Java
Implementation Code Location: ./Client/src/java
Package Name: com.iona
Select Service: GoogleSearchService
Select Port: GoogleSearchPort

Figure 7: *Development Tool, Showing Java Code Generation Options*

Deployment Tool

The Deployment Tool is invoked by selecting the **Deployment** icon under one of the services in the project tree. The Deployment Tool, shown in Figure 8, generates an Artix configuration file that is optimized for the selected service, a script for setting up your Artix runtime environment, and a composite Artix contract that is suitable for deployment into a runtime system. The generated configuration file contains all of the information needed to deploy your service using Artix. In the case of a standalone Artix service the Deployment Tool also generates start and stop scripts for the Artix service.



Current View: All

Domain name: Client
File location: .\Client\src/ClientCfg.zip
Logging Output: Standard Output
Logging Level: Errors Only

Figure 8: *Deployment Tool*

CHAPTER 2

Artix Java in Action: A Demo

The Artix product kit contains a set of demos showing Artix Java in action with different message and transport formats. This chapter walks you through one of those demos, and then explains how build an Artix Java application using your own WSDL file.

In this chapter

This chapter discusses the following topics:

The Hello World Demo

Overview

The Hello World demo is one that is widely used to showcase the capabilities of the Artix product suite. Depending on the Artix feature you are trying to explore, you can use the Hello World demo to focus on specific features of the product.

In this section, you will create a Java-based server and code it using the Artix Java APIs which will be configured to receive SOAP over HTTP requests sent by a C++-based client.

The server is coded independently of the transport and message formats using the Artix Java APIs. The Artix Java configuration method defines how the messages and transports are decoded. There are no hard-coded references

The Artix Java code is identical for every message and/or transport format used by the developer.

Setting the Environment

To set up the environment so that you can run this demo, you need to ensure that you have sourced artix_env as follows:

Windows

```
> %IT_PRODUCT_DIR%\artix\1.2\contrib\jArtix\bin\jArtix_env.bat
```

UNIX

```
% . $IT_PRODUCT_DIR/artix/1.2/contrib/jArtix/bin/jartix_env
```

This will set up your environment so that you can develop and run Artix applications. It also ensures that your classpath contains all required Artix Java JAR files.

Compiling and Running the Java Server

This demo contains a server that has two methods: greetMe, and sayHi, which both return text strings. The sayHi method is a simple string which returns the text "hi", while the greetMe method allows you to customize the text which is returned to the client.

To compile and run the server:

1. Go to the server_http directory
2. Compile the Java files:

```
javac *.java
```

3. Run the Java server

Windows

```
> start java Server
```

UNIX

```
% java server &
```

Now you're ready to compile and run the client.

Compiling and Running the C++ Client

The client will invoke on the two methods defined by the server's WSDL contract (sayHi and greetMe). To compile and run the client:

1. Move to the demos/hello_world/http_soap/client directory.
2. Compile the client:

Windows

```
> nmake -e all
```

UNIX

```
% make -e all
```

3. Run the client application:

Windows

```
> client.exe
```

UNIX

```
% ./client
```

The server will display messages received from the C++ client, and the client will show the responses received. If you wish, the client can be invoked with a command line parameter and this will be passed through in the greetMe method, for example:

Windows

```
> client.exe joe
```

UNIX

```
% ./client joe
```

The greetMe message will return with joe in the response text.

Building an Artix Java Server

Overview

This section will guide you through taking an existing WSDL file and use the Artix WSDLToJava application to generate a Java server which implements the WSDL interface.

Before you begin

It is assumed that you have installed Artix, and that you have a valid WSDL file. A sample WSDL file (helloworld.wsdl) is stored in the product kit within the HTTP_SOAP directory.

Setting the Environment

To set up the environment so that you can run this demo, you need to ensure that you have sourced artix_env as follows:

Windows

```
> %IT_PRODUCT_DIR%\artix\1.2\contrib\jArtix\bin\jArtix_env.bat
```

UNIX

```
% . $IT_PRODUCT_DIR/artix/1.2/contrib/jArtix/bin/jartix_env
```

Generating the Java Code

To generate the Java code:

1. Go to the directory that contains your WSDL file.
2. Run the WSDL conversion tool, specifying the WSDL file to be converted. For the purpose of this example, the HelloWorld.WSDL file is used:

Windows

```
> %IT_PRODUCT_DIR%\artix\1.2\contrib\jArtix\bin\wsdltojava
    helloworld.wsdl
```

UNIX

```
% . $IT_PRODUCT_DIR/artix/1.2/contrib/jArtix/bin/wsdltojava
    helloworld.wsdl
```

3. The following files are generated:

```
HelloWorldImpl.Java
HelloWorldServer.Java
```

Editing and Compiling the Code

Edit the implementation logic

You now need to write the implementation logic associated with this server.

1. Edit the HelloWorldImpl.Java file to change the greetMe and sayHi operations as shown here:

Example 2: *HelloWorldImpl.Java file showing the edits required*

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

/**
 * HelloWorldImpl
 */
public class HelloWorldImpl {

    /**
     * greetMe
     *
     * @param: stringParam0 (String)
     * @return: String
     */
    public String greetMe(String stringParam0) {
      String greeting = "Hello " + stringParam0;
      return greeting;
    }
    /**
     * sayHi
     *
     * @return: String
     */
    public String sayHi() {
      String hi = "Hi Java for Artix User";
       return hi;
      }
    }
```

2. Create a server mainline.

Example 3: *Server mainline code*

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

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

        QName name = new
   QName("http://xmlbus.com/HelloWorld","HelloWorldService");
        ServerFactoryBase factory = new
   SingleInstanceFactory("./HelloWorld.wsdl", new
   HelloWorldImpl());
       Bus.registerServerFactory(name,factory,"HelloWorldPort");

        System.out.println ("Demo Server starting...");
        Bus.init(args);
        Bus.run();
    }
}
```

Compile and run the server

1. Use the Java compiler to compile the source files:

```
javac *.java
```

2. Run the server:

```
java HelloWorld Server
```

3. The Server console displays the message "Demo Server starting..."

Testing the Server

Start the server

Windows

```
> start java Server
```

UNIX

```
% java server &
```

Now you're ready to compile and run the client.

Running the client

The client will invoke on the two methods defined by the server's WSDL contract (sayHi and greetMe). To compile and run the client:

1. Move to the demos/hello_world/http_soap/client directory.
2. Compile the client:

Windows

```
> nmake -e all
```

UNIX

```
% make -e all
```

3. Run the client application:

Windows

```
> client.exe
```

UNIX

```
% ./client
```

Glossary

A

Artix Designer
A suite of GUI tools for creating and deploying Artix integration solutions.

B

Binding
A binding associates a specific transport/protocol and data format with the operations defined in a `<portType>`.

Bus
See Service Bus

Bridge
A usage mode in which Artix is used to integrate applications using different payload formats.

C

Connection
An established communication link between any two Artix endpoints.

Contract
An Artix contract is a WSDL file that defines the interface and all connection-related information for that interface. A contract contains two components: logical and physical. The logical contract defines things that are independent of the underlying transport and wire format, and is specified in the `<portType>`, `<operation>`, `<message>`, `<type>`, and `<schema>` WSDL tags.

The physical contract defines the payload format, middleware transport, and service groupings, and the mappings between these things and portType ‘operations.’ The physical contract is specified in the `<port>`, `<binding>` and `<service>` WSDL tags.

Contract Editor
A GUI tool used for editing Artix contracts. It provides several wizards for adding services, transports, and bindings to an Artix contract.

D

Deployment Mode

One of two ways in which an Artix application can be deployed: Embedded and Standalone. An embedded-mode Artix application is linked with Artix-generated stubs and skeletons to connect client and server to the service bus. A standalone application runs as a separate process in the form of a daemon.

E

Embedded Mode

Operational mode in which an application creates a Service Access Point, 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.

End-point

The runtime deployment of one or more contracts, where one or more transports and its marshalling is defined, and at least one contract results in a generated stub or skeleton (thus an end-point can be compiled into an application). Contrast with Service.

H

Host

The network node on which a particular service resides.

M

Marshalling Format

A marshalling format controls the layout of a message to be delivered over a transport. A marshalling format is bound to a transport in the WSDL definition of a Port and its binding. A binding can also be specified in a logical contract portType, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

P

Payload Format

The on-the-wire structure of a message over a given transport. A payload format is associated with a port (transport) in the WSDL via the binding definition.

Protocol

A protocol is a transport whose format is defined by an open standard.

R

Routing
The redirection of a message from one WSDL binding to another. Routing rules are specified in a contract and apply to both end-points and standalone services. Artix supports port-based routing and operation-based routing defined in WSDL contracts. Content-based routing is supported at the application level.

Router
A usage mode in which Artix redirects messages based on rules defined in an Artix contract.

S

Service
An Artix service is an instance of an Artix runtime deployed with one or more contracts, but with no generated language bindings. The service has no compile-time dependencies. A service is dynamically configured by deploying one or more contracts on it.

Service Access Point
The mechanism, and the points at which individual service providers and consumers connect to the service bus.

Service Bus
The set of service providers and consumers that communicate via Artix. Also known as an Enterprise Service Bus.

Standalone Mode
An Artix instance running independently of either of the applications it is integrating. This provides a minimally invasive integration solution and is fully described by an Artix contract.

Switch
A usage mode in which Artix connects applications using two different transport mechanisms.

System
A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-In

A plug-in module that provides wire-level interoperation with a specific type of middleware. When configured with a given transport plug-in, Artix will interoperate with the specified middleware at a remote location or in another process. The transport is specified in the ‘Port’ property of a contract.

Index