Artix™

Developing Artix Applications in Java

Version 1.3, December 2003

Making Software Work Together™

Updated: 12-Dec-2003

M 3 1 8 3

Contents

Preface

Audience

This guide is intended for Artix Java programmers. In addition to a knowledge of Java, this guide assumes that the reader is familiar with WSDL and XML schemas.

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.

The **Help** menu in Artix Designer provides access to this online help.

Related documentation

The library for Artix includes the following:

- *Getting Started with Artix*
- *Artix Tutorial*
- *Deploying & Managing Artix Solutions*
- *Designing Artix Solutions*
- *Artix C++ Programmer's Guide*
- *Developing Artix Applications with Java*
- *Artix Security Guide*
- *Artix Thread Library Reference*

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

Reading path

If you are new to Artix, you should read the documentation in the following order:

1. *Getting Started with Artix*

   The getting started book describes the basic concepts behind Artix. It also provides details on installing the system and a detailed walk through for developing a C++ client for a Web Service.

2. *Artix Tutorial*

   The tutorial guides you through programming Artix applications against all of the supported transports.

3. *Deploying & Managing Artix Solutions*

   This guide provides details about the services and capabilities of Artix and how to deploy them into your software environment.

Help resources

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 `doc-feedback@iona.com`.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

`http://www.iona.com/support/kb/`

The IONA update center contains the latest releases and patches for IONA products:

`http://www.iona.com/support/update/`

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, Windows NT, Windows 95, or Windows 98 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

Developing Artix Enabled Clients and Servers

Artix generates stub and skeleton code that provides a developer with a simple model to develop transport independent applications.

In this chapter

This chapter discusses the following topics:

Generating Stub and Skeleton Code

Overview

The Artix development tools include a utility to generate server skeleton and client stub code from an Artix contract. The generated code is similar to code generated by a CORBA IDL compiler. There are two major differences between CORBA generated code and Artix generated code:

- Artix generated code is not restricted to using IIOP and therefore contains generic code that is compatible with a multitude of transports.
- Artix maps WSDL types to Java using the mapping described in the JAX-RPC specification. The resulting types are very different from those generated by an IDL-to-Java compiler.

Generated files

The Artix code generator produces a number of files from the Artix contract. They are named according to the port name from which the code was generated. The files include:

*PortName***.java** defines the Java interface that the client and server implement.

*PortName***Impl.java** defines the class used to implement the server.

*PortName***Server.java** is a simple main class for the server.

In addition to these files, the code generator also creates a class for each schema type or complex/simple type defined in the contract. These files are named according to the type name they are given in the contract and contain the helper functions needed to use the data types. The naming convention for the nelper type functions conforms to the JAX-RPC specification.

Generating code from the command line

You can generate code at the command line using the command:

```
wsdltojava [-e service] [-t port] [-b binding]  [-d output_dir]
   [-p package] [-impl] [-server] [-types] [-interface]
   [-sample] [-client] [-v] [-?] artix-contract
```

You must specify the location of a valid Artix contract for the code generator to work. The default behavior of `wsdltojava` is to generate all of the needed java code. You can also supply the following optional parameters to control the portions of the code generated:

`-e` *`service`*	Specifies the name of the service for which the tool will generate code. The default is to use the first service listed in the contract.
`-t` *`port`*	Specifies the name of the port for which code is generated. The default is to used the first port listed in the service.
`-b` *`binding`*	Specifies the name of the binding to use when generating code. The default is the first binding listed in the contract.
`-d` *`output_dir`*	Specifies the directory to which the generated code is written. The default is the current working directory.
`-p` *`package`*	Specifies the name of the Java package to use for the generated code.
`-impl`	Generates the skeleton class for implementing the server defined by the contract.
`-server`	Generates a simple main class for the server.
`-types`	Generates the code to implement the complex types defined by the contract.
`-interface`	Generates the Java interface for the service.
`-sample`	Generates a sample client that can be used to test your Java server.
`-client`	Generates only the Java interface and code needed to implement the complex types defined by the contract. This flag is equivalent to specifying `-interface -types`.
`-v`	Displays the version of the tool.
`-?`	Displays help on using the command line tool.

Java Package Names

Artix packages

The Artix Bus object which provides the transport and payload format independence in Artix is defined in the `com.iona.jbus` package. You will need to import this package and all of its subpackages for all Artix Java applications.

Java packages

Artix applications require a number of standard Java packages. These include:

javax.xml.namespace.QName provides the functionality to work with the XML QNames used to specifies services.

javax.xml.rpc.*. provides the APIs used to implement Artix Java clients. This package is not needed by server code.

java.io.* provides system input and output through data streams, serialization and the file system.

java.net.* provides the classes need to for communicating over a network. These classes are key to Artix applications that act as Web services.

Developing a Server

Overview

The Artix code generator generates server skeleton code and the implementation shell that serves as the starting point for developing a server that uses Artix. This skeleton code hides the transport details from the application developer, allowing them to focus on business logic.

Generating the server implementation class

The Artix code generator utility, `wsdltojava`, will generate an implementation class for your server when passed the `-impl` command flag.

Note: If your contract specifies any derived types or complex types you will also need to generate the code for supporting those types by specifying the `-types` flag.

Generated code

The implementation class code consists of two files:

PortName.**java** contains the interfaces that the server implements.

*PortName***Impl.java** contains the class definition for the server's implementation class. It also contains empty shells for the methods that implement the operations defined in the contract.

Completing the server implementation

You must provide the logic for the operations specified in the contract that defines the server. To do this you edit the empty methods provided in *`PortName`*`Impl.java`. A generated impl class for a contract defining a service with two operations, `sayHi` and `greetMe`, would resemble Example 1. Only the code portions highlighted in **`bold`** (in the bodies of the `greetMe()` and `sayHi()` methods) must be inserted by the programmer. Writing the server

Example 1: *Implementation of the HelloWorld PortType in the Server*

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

Example 1: *Implementation of the HelloWorld PortType in the Server*

```
public class HelloWorldImpl {

    /**
     * greetMe
     *
     * @param: stringParam0 (String)
     * @return: String
     */
   public String greetMe(String stringParam0) {
        System.out.println("HelloWorldSkel::greetMe called with
   message: "+stringParam0);
        return "Hello Artix User: "+stringParam0;
    }

    /**
     * sayHi
     *
     * @return: String
     */
    public String sayHi() {
        System.out.println("HelloWorldSkel::sayHi called");
        return "Greetings from the Artix HelloWorld Server";
    }
```

main()

The server `main()` of an Artix Java server must do three things before it can service requests:

1. Initialize the Artix bus.
2. Register a factory for the server implementation with the Artix bus.
3. Start the Artix bus.

You can use `wsdltojava` to generate a server `main()` with the code to perform these steps by using the `-server` flag. The main() shown in Example 4 on page 8 was generated using `wsdltojava`.

Initializing the bus

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

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

This will create a bus instance to host your services, load the Artix configuration information for your application, and load the required plug-ins.

Registering a factory for the server implementation

Before the bus can begin processing requests made on your server, you must register a factory for the object that implements the server's business logic with the bus. Registering a factory for the implementation object with the bus allows the bus to create instances of the implementation object to service requests.

To register a factory for your implementation object you create a `com.iona.jbus.ServerFactoryBase` using the path of the WSDL file describing the service interface and an instance of your implementation object. Example 2 shows the code to create a server factory for the `HelloWorld` service.

Example 2: *Creating a ServerFactoryBase*

```
//Java
ServerFactoryBase factory =
  new SingleInstanceFactory("./HelloWorld.wsdl",
                            new HelloWorldImpl());
```

After creating the server factory, you register it with the bus using the bus' `registerServerFactory()` method. The signature for `registerServerFactory()` is shown in Example 3.

Example 3: *registerServerFactory()*

```
void registerServerFactory(QName serviceName,
                           ServerFactoryBase factory,
                           String portName)
throws BusException
```

In addition to the server factory, `registerServerFactory()` takes the service's QName as specified in the contract defining the service and the name of the WSDL port the service is instantiating.

Starting the bus

After the bus is initialized and the server implementation is registered with it, the bus is ready to listen for requests and pass them to the server for processing. To start the bus, you use the bus' `run()` method. Once the bus is started, it retains control of the process until it is shut down. The server's `main()` will be blocked until `run()` returns.

Completed server main()

Example 4 on page 8 shows how the `main()` for a Java Artix server might look.

Example 4: *Server main()*

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

public class Server
{
  public static void main(String args[])
  throws Exception
  {
    // Initialize the Artix bus
    Bus bus = Bus.init(args);

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

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

Developing a Client

Overview

Artix Java clients are implemented using dynamic proxies as described in the JAX-RPC 1.1 specification. The interface used to create the proxy class is defined in the generated file *PortName*.**java**. The only Artix specific code that Artix Java clients need to use is to initialize and shutdown the Artix bus.

Initializing the bus

Client applications initialize the bus in the same manner as server applications, by calling the bus' `init()` method. Client applications, however, do not need to make a call to the bus' `run()` method.

Instantiating the client proxy

Artix Java clients use dynamic proxies, as described in the JAX-RPC specification, to make requests on servers. Dynamic proxies are created using the interface generated from your contract and the `javax.xml.rpc.Service` interface. You need the `QName` of the service for which you are creating the proxy, the `QName` of the endpoint with which the proxy will contact the service, and the URL of the contract defining the service. Once you have these three pieces of information, creating a dynamic proxy requires three steps:

1. Obtain an instance of `javax.xml.rpc.ServiceFactory` to create the service.
2. Use the `ServiceFactory` to create a `Service` instance for the service to which the proxy will connect.
3. Use the `Service` to instantiate the dynamic proxy.

Obtaining a ServiceFactory instance

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

Creating a Service instance

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

- the URL of the contract defining the service.

- the service's `QName`.

Creating the dynamic proxy

The dynamic proxy is created from the `Service` using `getPort()`. `getPort()` takes two arguments:

- the `QName` of the endpoint with which the proxy contacts the service.
- the name of the generated Java interface in *`PortName`*`.java` with `.class` appended. For example, if the generated interface's name is `HelloWorld`, this argument would be `HelloWorld.class`.

`getPort()` returns an instance of `java.rmi.Remote` that must be cast to the generated interface.

Shutting the bus down

Unlike a server that must shut down the bus from a separate thread, clients do not typically make a call to the bus' `run()` method and can simply call `shutdown()` on the bus before the main thread exits. It is advisable to pass `true` to `shutdown()` to ensure that the bus is fully shutdown before exiting.

Full client code

An Artix Java client developed to access the service `HelloWorldService` will look similar to Example 5.

Example 5: *Client Code*

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

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;

public class HelloWorldClient
{

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

2   QName name = new QName("http://xmlbus.com/HelloWorld",
                           "HelloWorldService");
```

Example 5: *Client Code*

```
3       QName portName = new QName("","HelloWorldPort");

4       String wsdlPath =
        "file:/C:/IONA/artix/1.3/demos/java/hello_world/http_soap/Hel
        loWorld.wsdl";
        URL wsdlLocation = new File(wsdlPath).toURL();

5       ServiceFactory factory = ServiceFactory.newInstance();

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

7       HelloWorld impl = (HelloWorld)service.getPort(portName,
                                                HelloWorld.class);

8       String string_out;

        string_out = impl.sayHi();
        System.out.println(string_out);

9       bus.shutdown(true);

        }
}
```

The code does the following:

1. The `com.iona.jbus.Bus.init()` function initializes the bus.
2. Creates the service's `QName`.
3. Creates the `QName` of the endpoint with which the proxy will contact the service.
4. Creates the URL of the contract defining the service.
5. The `newInstance()` function returns the `ServiceFactory`.
6. The `createService()` function instantiates the `Service` from which the dynamic proxy is created.
7. The `getPort()` function returns a dynamic proxy to the `HelloWorld` service. `getPort()` returns an instance of `java.rmi.Remote` that must be cast to the interface defining the service.
8. Make a call on the proxy to request service.
9. Shutdown the bus.

Building an Artix Application

Required jar files

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

- `it_bus.jar`
- `it_wsdl.jar`
- `it_ws_reflect.jar`
- `ifc.jar`

You also need to ensure that the Artix version of *`jaxrpc-api.jar`* is used to build your Artix application. The simplest way to make sure the correct version is used is to prepend *`artix_install_dir`*`\artix\1.3\lib` to your class path.

Index